



Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



Introduction to Java

Welcome to the exciting world of Java! If you're new to programming, you might be wondering what Java is, why it's so popular, and how it can help you create amazing things like apps, games, or even websites. Don't worry if you've never written a line of code before—this blog is just for you! Let's dive into the basics of Java in a simple and friendly way.

What is Java?

Java is a high level class based object-oriented programming language that lets you tell a computer what to do. Think of it as a set of instructions you give to your computer, kind of like a recipe for your favorite dish. Java was created in 1995 by a company called Sun Microsystems (now part of Oracle), and it's been a favorite among developers ever since. Why? Because it's versatile, reliable, and works on almost any device—your phone, laptop, or even a smart fridge!

Why Learn Java?

You might be asking, "Why should I start with Java instead of another language?" Great question! Here are a few reasons why Java is perfect for beginners:

1. **It's Everywhere:** Java powers everything from Android apps to banking systems. If you learn Java, you're learning a skill that's in high demand.



2. **Beginner-Friendly:** Java's rules (called syntax) are straightforward, making it easier to understand than some other languages.
3. **“Write Once, Run Anywhere”:** Java works on any device with a Java Virtual Machine (JVM). You write your code once, and it can run on Windows, Mac, Linux—you name it!
4. **Lots of Help:** Java has a huge community of learners and experts, so if you get stuck, there's always someone or something (like this blog!) to guide you.

How Does Java Work?

Let's break it down simply. When you write Java code, you're giving the computer a list of tasks. But computers don't understand English—they speak in 1s and 0s (called machine language). Java acts like a translator:

1. You write your code in Java (human-readable).
2. The Java Compiler (`javac`) turns it into something called “bytecode.”
3. The Java Virtual Machine (JVM) takes that bytecode and runs it on your computer, no matter what type it is.

Think of it like this: You're a chef writing a recipe (Java code), the compiler translates it into a universal cooking language (bytecode), and the JVM is the kitchen that cooks it perfectly on any stove!

Your First Java Program

Let's write a super simple program to say “Hello, World!” This is a tradition in programming—it's like your first “Hi” to the coding world.

Here's the code:



```
public class Main {  
    public static void main(String[] args) {  
        System.out.println("Hello, World!");  
    }  
}
```

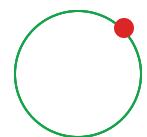


Don't panic if this looks confusing! Let's break it down:

- `public class Main`: This is like naming your recipe “Main.” Every Java program needs a class (a container for your code).
- `public static void main`: This is the starting point of your program—like pressing “Play” on a video. Java looks for this to know where to begin.
- `System.out.println("Hello, World!");`: This tells the computer to print “Hello, World!” on the screen. The semicolon (;) is like a period at the end of a sentence.

To run this:

1. Open any Online Java Compiler or you can use Coding Shuttle’s Online Java Compiler
<https://www.codingshuttle.com/compilers/java/>
2. Copy and Paste the Above code and simply click on Run button, you will see the output Hello World Printed in the Output section.



The screenshot shows the CodingShuttle Online Java Compiler interface. On the left, there's a sidebar with icons for Java, C, C++, Python, JavaScript, Go, and C#. The main area has tabs for "Java Online Compiler" and "C Online Compiler". A code editor window displays the following Java code:

```
1 public class Main {  
2     public static void main(String[] args) {  
3         System.out.println("Hello, World!");  
4     }  
5 }
```

Below the code are buttons for "Share", "Settings", "File", and "Run". The "Run" button is highlighted in purple. To the right, the "Output" section shows the status "Accepted", memory usage "9896 byte", and execution time "0.083 sec". The output text "Hello, World!" is displayed. Below the output is an "Input" section with a placeholder "Enter your input here...".

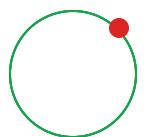
CodingShuttle Online Java Compiler

Tips for Learning Java

- **Start Small:** Don't try to build a game on day one. Begin with simple programs like the one above.
- **Practice:** Write code every day, even if it's just a few lines.
- **Ask Questions:** If something doesn't make sense, search online or ask a friend. Mistakes are part of learning!
- **Have Fun:** Programming is like solving puzzles—enjoy the process.

What's Next?

Now that you've got a taste of Java, you can explore more! Try changing “Hello, World!” to your name or adding two numbers together. In future sections of your Java Handbook, we'll cover topics like loops, conditions, and all the important topics that you must know.





 Pay Day Sale is Now Live!  Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



Download and Install JDK on your Computer

In this blog I will guide you to setup java on your computer so that you can run your first java program using Notepad.

What is JDK and Why it is required?

For Developing Java Program we require JDK (Java Development Kit), It will have compiler and the Runtime environment which will help us to write a java program and execute them.

Once you install JDK along with that you will get two more things that is JRE (Java Runtime Environment) and JVM (Java Virtual Machine).





JDK, JRE and JVM

Let's understand each of this one by one

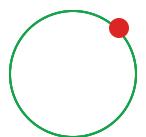
1. JDK - JDK is Java Development Kit it will have all the development tools required for compiling, debugging the java program, and various other tools are available.
2. JRE - Once the program is compiled you have to execute it, for that we require JRE. JRE contains the library of the Java Classes and JVM
3. JVM - JVM comes with JRE and it is the one which actually executes our program.

How all this work together?

Let's say we have a Java file with name **First.java**, when we compile this file with **javac First.** **java** we will get **First.class** a compiled file.

Now to execute **First.class** file we have to simply run **java First** command, it will start executing the file.

So, for compilation we use **javac** and for execution we use **java**.



Download and Install JDK

After understanding JDK, JRE and JVM it's time to download and install JDK on your computer and compile and execute your first java program.

1. Visit :- <https://www.oracle.com/java/technologies/downloads/#jdk24-windows> and install the latest version of jdk, I will install JDK 24 x64 MSI Installer you can choose other options also.

The screenshot shows the Oracle Java Downloads page. At the top, there is a navigation bar with links for Apps, Google, Content Kanban Board, aws home, Super Set, Lucid App, Excalidraw-thumbn..., Content Manager, Meet - baw-zezd-ufy, Our Blogs | Coding..., and All Bookmarks. Below the navigation bar, the Oracle logo is visible along with links for Products, Industries, Resources, Customers, Partners, Developers, and Company. A search bar and a language selection dropdown (American English) are also present. The main content area is titled "Java SE Development Kit 24 downloads". It states that JDK 24 is the latest Long-Term Support (LTS) release of the Java SE Platform. Below this, it says "Earlier JDK versions are available below." A horizontal menu bar at the bottom of this section includes links for JDK 24, JDK 23, JDK 21, GraalVM for JDK 24, GraalVM for JDK 23, and GraalVM for JDK 21. The "Java 24" link is highlighted. The "Windows" tab is selected under the "Java SE Development Kit 24 downloads" section. A table lists three download options:

Product/file description	File size	Download
x64 Compressed Archive	229.40 MB	https://download.oracle.com/java/24/latest/jdk-24_windows-x64_bin.zip (sha256)
x64 Installer	205.76 MB	https://download.oracle.com/java/24/latest/jdk-24_windows-x64_bin.exe (sha256)
x64 MSI Installer	204.51 MB	https://download.oracle.com/java/24/latest/jdk-24_windows-x64_bin.msi (sha256)

Oracle Website

1. Once downloaded simply click on installer file (.exe or .msi or .dmg) and install the setup
2. After installation completes open terminal and verify the java version by running `java --version` command, you will see the java version which you have installed.

The screenshot shows a terminal window with a dark background. The command `java --version` is entered and its output is displayed:
C:\\\\Users\\\\Prem Mane>java --version
java 24 2025-03-18
Java(TM) SE Runtime Environment (build 24+36-3646)
Java HotSpot(TM) 64-Bit Server VM (build 24+36-3646, mixed mode, sharing)
C:\\\\Users\\\\Prem Mane>

Compile and Execute your first java program

Now you have successfully setup a Java Environment on your machine. It's time to write your first hello world program and run it.

1. Create a new file and paste the following code in the file and save the file with name First and extension .java (First.java).

```
public class First {  
    public static void main(String[] args) {  
        System.out.println("Hello, World!");  
    }  
}
```



1. After that open command prompt and go to the the location where you have saved the file.
2. Run this command `javac [First.java](<http://First.java>)` to compile the file, this will create First.class (Compiled file).
3. Now we have to simply execute this compiled file and to execute run this command `java First`

Output:

```
C:\\\\Users\\\\Prem Mane\\\\OneDrive\\\\Desktop\\\\Java Programs>dir  
Volume in drive C is OS  
Volume Serial Number is 42A8-7BC9  
  
Directory of C:\\\\Users\\\\Prem Mane\\\\OneDrive\\\\Desktop\\\\Java Programs  
  
01-04-2025  15:09      <DIR>          .  
01-04-2025  15:08      <DIR>          ..  
01-04-2025  15:11                  122 First.java  
                           1 File(s)           122 bytes
```





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Java Comments

Imagine you've written a Java program that you need to share with your project team. Without comments, your teammates might struggle to understand what your code does. To make your code more understandable, you should add comments explaining its functionality.

Comments are of two types

1. Single-line Comment

In Java, a single-line comment starts and ends in the same line. To write a single-line comment, we can use the `//` symbol. For example,

```
class HelloWorld {  
    public static void main(String[] args) {  
        // print Hello World to the screen  
        System.out.println("Hello World");  
    }  
}
```

2. Multi-line Comment

When we want to write comments in multiple lines, we can use the multi-line comment. To write multi-line comments, we can use the `/*....*/` symbol. For example,





 Pay Day Sale is Now Live!  Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



 Java Programming Handbook



Java is one of the most widely used and versatile programming languages in the world. From desktop applications to web services and mobile apps to enterprise systems, Java powers countless technologies and platforms.

This Java tutorial is designed for beginners who want to build a strong foundation in Java programming. Whether you're completely new to programming or transitioning from another language, this guide covers everything from core concepts to essential object-oriented principles and beyond.

If you're aiming to become a proficient Java developer, follow this structured path to master Java, step by step.

Prerequisites

Before you dive into Java, it's helpful (but not mandatory) to be familiar with:

- **Basic Computer Knowledge:** Understanding how software runs, what an IDE is, etc.
- **Logic Building:** Basic problem-solving and logical thinking.



- **Understanding of Programming Concepts** (optional): Knowing variables, loops, or functions from another language is a bonus.

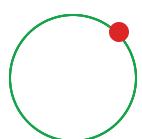
Master These Topics

Once you're ready, follow this structured list to learn Java from the ground up:

1. Introduction to Java
2. Java Fundamentals
3. Java Flow Control
4. Java Arrays
5. Java Methods
6. Java Object-Oriented Programming
7. Java Inner Classes
8. Static and Final Keywords
9. Java Exception Handling
10. Java Generics
11. Java Lambda Expressions
12. Java Collection Framework
13. Java Input/Output Streams
14. Java Reader/Writer

Introduction to Java

1. [Introduction to Java](#)
2. [Installing JDK on your computer](#)



[3. Java Comments](#)

Java Fundamentals

- [1. Java Variables and Literals](#)
- [2. Data Types in Java](#)
- [3. Operators in Java](#)
- [4. Java Basic Input and Output](#)
- [5. Java Expressions, Statements and Blocks](#)

Java Flow Control

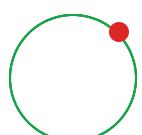
- [1. Java if else statement](#)
- [2. Java Ternary Operator](#)
- [3. Java For Loop](#)
- [4. Java while and do while loop](#)
- [5. Java continue and break statement](#)
- [6. Java Switch statement](#)

Java Arrays

- [1. Java Arrays](#)
- [2. Java Multidimensional Arrays](#)
- [3. Java Copy Arrays](#)

Java Methods

- [1. Methods in Java](#)



2. Parameter Passing in Java
3. Method Overloading
4. Variable Arguments
5. Java CommandLine Arguments
6. Java Recursive Method

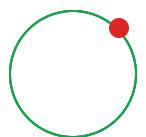
Java Object-Oriented Programming

1. Principles of Object-Oriented Programming
2. Java Class and Objects
3. Java Constructor
4. Encapsulation and Data Hiding
5. Inheritance in Java
6. Constructors in Inheritance
7. this Vs super Keyword
8. Method Overriding
9. Dynamic Method Dispatch
10. Polymorphism using Overloading and Overriding
11. Abstract Classes
12. Interfaces in Java

Java Inner Classes

1. Inner Classes in Java

Static and Final Keywords



1. Static Members and Static Blocks in Java
2. Final Members
3. Singleton Class

Java Exception Handling

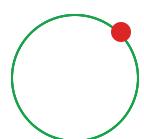
1. What are Exceptions
2. How to handle Exception
3. Try and Catch Block
4. Multiple and Nested Try Catch
5. Class Exception
6. Checked and Unchecked Exceptions
7. Throw Vs Throws
8. Finally Block
9. Try with Resources

Java Generics

1. Introduction to Generics
2. Defining Generic Class
3. Bounds on Generics
4. Java Generic Methods

Java Lambda Expressions

1. Introduction to Lambda Expressions
2. Parameters in Lambda Expressions



3. Capture in Lambda Expression

4. Method References

Java Collection Framework

1. Collections in Java

2. Understanding Collection Interface

3. Understanding List Interface in Java

4. Java ArrayList

5. Java LinkedList

6. Java Vector

7. Java Stack

8. Understanding Queue Interface in Java

9. Java PriorityQueue

10. Java ArrayDeque

11. Understanding Set Interface in Java

12. Java HashSet

13. Java LinkedHashSet

14. Java TreeSet

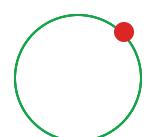
15. Understanding Map Interface in Java

16. Java HashMap

17. Java LinkedHashMap

18. Java TreeMap

19. Comparable Interface in Java



20. Comparator Interface in Java

Java Input/Output Streams

1. [What are Streams](#)
2. [InputStream and OutputStream Classes](#)
3. [FileInputStream and FileOutputStream](#)
4. [ByteArrayInputStream and ByteArrayOutputStream](#)
5. [ObjectInputStream and ObjectOutputStream](#)
6. [BufferedInputStream and BufferedOutputStream](#)
7. [Java PrintStream Class](#)

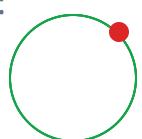
Java Reader/Writer

1. [Java File Class](#)
2. [Java Reader and Writer Class](#)
3. [InputStreamReader and OutputStreamWriter Class](#)
4. [FileReader and FileWriter Class](#)
5. [BufferedReader and BufferedWriter Class](#)
6. [StringReader and StringWriter Class](#)
7. [Java PrintWriter Class](#)

Key Features of Java

Java has remained popular for decades for good reason. Here's what makes it special:

- **Platform Independent:** Write once, run anywhere (thanks to the JVM).



Java Tutorial – A Comprehensive Guide for Beginners

Introduction

Java is one of the most widely used and versatile programming languages in the world. From desktop applications to web services and mobile apps to enterprise systems, Java powers countless technologies and platforms.

This Java tutorial is designed for beginners who want to build a strong foundation in Java programming. Whether you're completely new to programming or transitioning from another language, this guide covers everything from core concepts to essential object-oriented



Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#).

A rounded rectangular navigation bar with a light gray background. It features three icons: a menu icon (three horizontal lines), a search icon (magnifying glass), and a back/forward icon (up arrow). The text "Java Programming Handbook" is centered in the middle of the bar.

Before you dive into Java, it's helpful (but not mandatory) to be familiar with:

- **Basic Computer Knowledge:** Understanding how software runs, what an IDE is, etc.
- **Logic Building:** Basic problem-solving and logical thinking.



- **Understanding of Programming Concepts** (optional): Knowing variables, loops, or functions from another language is a bonus.

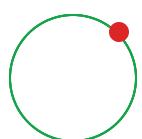
Master These Topics

Once you're ready, follow this structured list to learn Java from the ground up:

1. Introduction to Java
2. Java Fundamentals
3. Java Flow Control
4. Java Arrays
5. Java Methods
6. Java Object-Oriented Programming
7. Java Inner Classes
8. Static and Final Keywords
9. Java Exception Handling
10. Java Generics
11. Java Lambda Expressions
12. Java Collection Framework
13. Java Input/Output Streams
14. Java Reader/Writer

Introduction to Java

1. [Introduction to Java](#)
2. [Installing JDK on your computer](#)



[3. Java Comments](#)

Java Fundamentals

- [1. Java Variables and Literals](#)
- [2. Data Types in Java](#)
- [3. Operators in Java](#)
- [4. Java Basic Input and Output](#)
- [5. Java Expressions, Statements and Blocks](#)

Java Flow Control

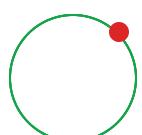
- [1. Java if else statement](#)
- [2. Java Ternary Operator](#)
- [3. Java For Loop](#)
- [4. Java while and do while loop](#)
- [5. Java continue and break statement](#)
- [6. Java Switch statement](#)

Java Arrays

- [1. Java Arrays](#)
- [2. Java Multidimensional Arrays](#)
- [3. Java Copy Arrays](#)

Java Methods

- [1. Methods in Java](#)



2. Parameter Passing in Java
3. Method Overloading
4. Variable Arguments
5. Java CommandLine Arguments
6. Java Recursive Method

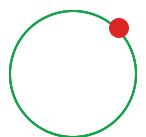
Java Object-Oriented Programming

1. Principles of Object-Oriented Programming
2. Java Class and Objects
3. Java Constructor
4. Encapsulation and Data Hiding
5. Inheritance in Java
6. Constructors in Inheritance
7. this Vs super Keyword
8. Method Overriding
9. Dynamic Method Dispatch
10. Polymorphism using Overloading and Overriding
11. Abstract Classes
12. Interfaces in Java

Java Inner Classes

1. Inner Classes in Java

Static and Final Keywords



1. Static Members and Static Blocks in Java
2. Final Members
3. Singleton Class

Java Exception Handling

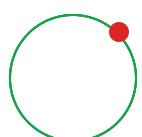
1. What are Exceptions
2. How to handle Exception
3. Try and Catch Block
4. Multiple and Nested Try Catch
5. Class Exception
6. Checked and Unchecked Exceptions
7. Throw Vs Throws
8. Finally Block
9. Try with Resources

Java Generics

1. Introduction to Generics
2. Defining Generic Class
3. Bounds on Generics
4. Java Generic Methods

Java Lambda Expressions

1. Introduction to Lambda Expressions
2. Parameters in Lambda Expressions



3. Capture in Lambda Expression

4. Method References

Java Collection Framework

1. Collections in Java

2. Understanding Collection Interface

3. Understanding List Interface in Java

4. Java ArrayList

5. Java LinkedList

6. Java Vector

7. Java Stack

8. Understanding Queue Interface in Java

9. Java PriorityQueue

10. Java ArrayDeque

11. Understanding Set Interface in Java

12. Java HashSet

13. Java LinkedHashSet

14. Java TreeSet

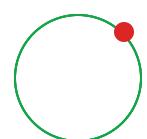
15. Understanding Map Interface in Java

16. Java HashMap

17. Java LinkedHashMap

18. Java TreeMap

19. Comparable Interface in Java



20. Comparator Interface in Java

Java Input/Output Streams

1. [What are Streams](#)
2. [InputStream and OutputStream Classes](#)
3. [FileInputStream and FileOutputStream](#)
4. [ByteArrayInputStream and ByteArrayOutputStream](#)
5. [ObjectInputStream and ObjectOutputStream](#)
6. [BufferedInputStream and BufferedOutputStream](#)
7. [Java PrintStream Class](#)

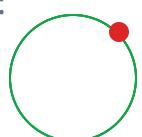
Java Reader/Writer

1. [Java File Class](#)
2. [Java Reader and Writer Class](#)
3. [InputStreamReader and OutputStreamWriter Class](#)
4. [FileReader and FileWriter Class](#)
5. [BufferedReader and BufferedWriter Class](#)
6. [StringReader and StringWriter Class](#)
7. [Java PrintWriter Class](#)

Key Features of Java

Java has remained popular for decades for good reason. Here's what makes it special:

- **Platform Independent:** Write once, run anywhere (thanks to the JVM).



Java Tutorial – A Comprehensive Guide for Beginners

Introduction

Java is one of the most widely used and versatile programming languages in the world. From desktop applications to web services and mobile apps to enterprise systems, Java powers countless technologies and platforms.

This Java tutorial is designed for beginners who want to build a strong foundation in Java programming. Whether you're completely new to programming or transitioning from another language, this guide covers everything from core concepts to essential object-oriented principles and beyond.

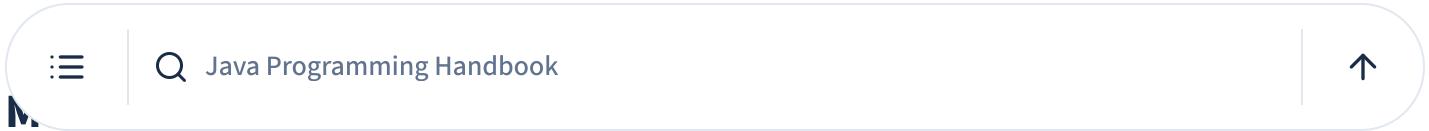
If you're aiming to become a proficient Java developer, follow this structured path to master Java, step by step.

Prerequisites

Before you dive into Java, it's helpful (but not mandatory) to be familiar with:

- **Basic Computer Knowledge:** Understanding how software runs, what an IDE is, etc.

 Pay Day Sale is Now Live!  Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#).

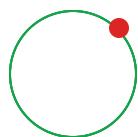


Once you're ready, follow this structured list to learn Java from the ground up:

1. Introduction to Java
2. Java Fundamentals
3. Java Flow Control
4. Java Arrays
5. Java Methods
6. Java Object-Oriented Programming
7. Java Inner Classes
8. Static and Final Keywords
9. Java Exception Handling
10. Java Generics
11. Java Lambda Expressions
12. Java Collection Framework
13. Java Input/Output Streams
14. Java Reader/Writer

Introduction to Java

1. [**Introduction to Java**](#)
2. [**Installing JDK on your computer**](#)



[3. Java Comments](#)

Java Fundamentals

- [1. Java Variables and Literals](#)
- [2. Data Types in Java](#)
- [3. Operators in Java](#)
- [4. Java Basic Input and Output](#)
- [5. Java Expressions, Statements and Blocks](#)

Java Flow Control

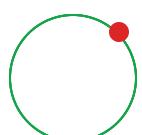
- [1. Java if else statement](#)
- [2. Java Ternary Operator](#)
- [3. Java For Loop](#)
- [4. Java while and do while loop](#)
- [5. Java continue and break statement](#)
- [6. Java Switch statement](#)

Java Arrays

- [1. Java Arrays](#)
- [2. Java Multidimensional Arrays](#)
- [3. Java Copy Arrays](#)

Java Methods

- [1. Methods in Java](#)



2. Parameter Passing in Java
3. Method Overloading
4. Variable Arguments
5. Java CommandLine Arguments
6. Java Recursive Method

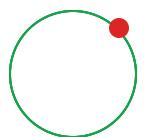
Java Object-Oriented Programming

1. Principles of Object-Oriented Programming
2. Java Class and Objects
3. Java Constructor
4. Encapsulation and Data Hiding
5. Inheritance in Java
6. Constructors in Inheritance
7. this Vs super Keyword
8. Method Overriding
9. Dynamic Method Dispatch
10. Polymorphism using Overloading and Overriding
11. Abstract Classes
12. Interfaces in Java

Java Inner Classes

1. Inner Classes in Java

Static and Final Keywords



1. Static Members and Static Blocks in Java
2. Final Members
3. Singleton Class

Java Exception Handling

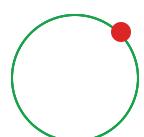
1. What are Exceptions
2. How to handle Exception
3. Try and Catch Block
4. Multiple and Nested Try Catch
5. Class Exception
6. Checked and Unchecked Exceptions
7. Throw Vs Throws
8. Finally Block
9. Try with Resources

Java Generics

1. Introduction to Generics
2. Defining Generic Class
3. Bounds on Generics
4. Java Generic Methods

Java Lambda Expressions

1. Introduction to Lambda Expressions
2. Parameters in Lambda Expressions



3. Capture in Lambda Expression

4. Method References

Java Collection Framework

1. Collections in Java

2. Understanding Collection Interface

3. Understanding List Interface in Java

4. Java ArrayList

5. Java LinkedList

6. Java Vector

7. Java Stack

8. Understanding Queue Interface in Java

9. Java PriorityQueue

10. Java ArrayDeque

11. Understanding Set Interface in Java

12. Java HashSet

13. Java LinkedHashSet

14. Java TreeSet

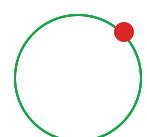
15. Understanding Map Interface in Java

16. Java HashMap

17. Java LinkedHashMap

18. Java TreeMap

19. Comparable Interface in Java



20. Comparator Interface in Java

Java Input/Output Streams

1. [What are Streams](#)
2. [InputStream and OutputStream Classes](#)
3. [FileInputStream and FileOutputStream](#)
4. [ByteArrayInputStream and ByteArrayOutputStream](#)
5. [ObjectInputStream and ObjectOutputStream](#)
6. [BufferedInputStream and BufferedOutputStream](#)
7. [Java PrintStream Class](#)

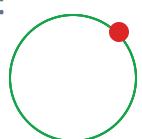
Java Reader/Writer

1. [Java File Class](#)
2. [Java Reader and Writer Class](#)
3. [InputStreamReader and OutputStreamWriter Class](#)
4. [FileReader and FileWriter Class](#)
5. [BufferedReader and BufferedWriter Class](#)
6. [StringReader and StringWriter Class](#)
7. [Java PrintWriter Class](#)

Key Features of Java

Java has remained popular for decades for good reason. Here's what makes it special:

- **Platform Independent:** Write once, run anywhere (thanks to the JVM).



Java Tutorial – A Comprehensive Guide for Beginners

Introduction

Java is one of the most widely used and versatile programming languages in the world. From desktop applications to web services and mobile apps to enterprise systems, Java powers countless technologies and platforms.

This Java tutorial is designed for beginners who want to build a strong foundation in Java programming. Whether you're completely new to programming or transitioning from another language, this guide covers everything from core concepts to essential object-oriented principles and beyond.

If you're aiming to become a proficient Java developer, follow this structured path to master Java, step by step.

Prerequisites

Before you dive into Java, it's helpful (but not mandatory) to be familiar with:

- **Basic Computer Knowledge:** Understanding how software runs, what an IDE is, etc.
- **Logic Building:** Basic problem-solving and logical thinking.



- **Understanding of Programming Concepts** (optional): Knowing variables, loops, or functions from another language is a bonus.

Master These Topics

Once you're ready, follow this structured list to learn Java from the ground up:

1. Introduction to Java
2. Java Fundamentals
3. Java Flow Control
4. Java Arrays
5. Java Methods
6. Java Object-Oriented Programming
7. Java Inner Classes
8. Static and Final Keywords
9. Java Exception Handling
10. Java Generics
11. Java Lambda Expressions



Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)

14. Java Reader/Writer

The screenshot shows a mobile application interface for learning Java. At the top, there's a navigation bar with three icons: a menu, a search, and a user profile. Below the bar, a teal banner announces a "Pay Day Sale" with a discount code. The main content area has a light gray background. On the left, a sidebar contains a list of topics: "Java Reader/Writer", "Java Programming Handbook", and "Java Syntax". The main content area displays a numbered list of Java topics. A large green circular icon with a red dot is visible in the bottom right corner.

1. [Introduction to Java](#)
2. [Installing JDK on your computer](#)

[3. Java Comments](#)

Java Fundamentals

- [1. Java Variables and Literals](#)
- [2. Data Types in Java](#)
- [3. Operators in Java](#)
- [4. Java Basic Input and Output](#)
- [5. Java Expressions, Statements and Blocks](#)

Java Flow Control

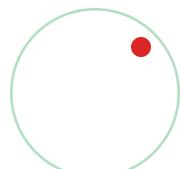
- [1. Java if else statement](#)
- [2. Java Ternary Operator](#)
- [3. Java For Loop](#)
- [4. Java while and do while loop](#)
- [5. Java continue and break statement](#)
- [6. Java Switch statement](#)

Java Arrays

- [1. Java Arrays](#)
- [2. Java Multidimensional Arrays](#)
- [3. Java Copy Arrays](#)

Java Methods

- [1. Methods in Java](#)



2. Parameter Passing in Java
3. Method Overloading
4. Variable Arguments
5. Java CommandLine Arguments
6. Java Recursive Method

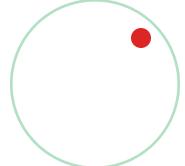
Java Object-Oriented Programming

1. Principles of Object-Oriented Programming
2. Java Class and Objects
3. Java Constructor
4. Encapsulation and Data Hiding
5. Inheritance in Java
6. Constructors in Inheritance
7. this Vs super Keyword
8. Method Overriding
9. Dynamic Method Dispatch
10. Polymorphism using Overloading and Overriding
11. Abstract Classes
12. Interfaces in Java

Java Inner Classes

1. Inner Classes in Java

Static and Final Keywords



1. Static Members and Static Blocks in Java
2. Final Members
3. Singleton Class

Java Exception Handling

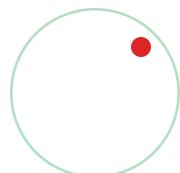
1. What are Exceptions
2. How to handle Exception
3. Try and Catch Block
4. Multiple and Nested Try Catch
5. Class Exception
6. Checked and Unchecked Exceptions
7. Throw Vs Throws
8. Finally Block
9. Try with Resources

Java Generics

1. Introduction to Generics
2. Defining Generic Class
3. Bounds on Generics
4. Java Generic Methods

Java Lambda Expressions

1. Introduction to Lambda Expressions
2. Parameters in Lambda Expressions



3. Capture in Lambda Expression

4. Method References

Java Collection Framework

1. Collections in Java

2. Understanding Collection Interface

3. Understanding List Interface in Java

4. Java ArrayList

5. Java LinkedList

6. Java Vector

7. Java Stack

8. Understanding Queue Interface in Java

9. Java PriorityQueue

10. Java ArrayDeque

11. Understanding Set Interface in Java

12. Java HashSet

13. Java LinkedHashSet

14. Java TreeSet

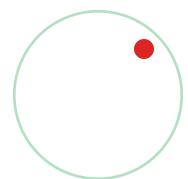
15. Understanding Map Interface in Java

16. Java HashMap

17. Java LinkedHashMap

18. Java TreeMap

19. Comparable Interface in Java



20. Comparator Interface in Java

Java Input/Output Streams

1. [What are Streams](#)
2. [InputStream and OutputStream Classes](#)
3. [FileInputStream and FileOutputStream](#)
4. [ByteArrayInputStream and ByteArrayOutputStream](#)
5. [ObjectInputStream and ObjectOutputStream](#)
6. [BufferedInputStream and BufferedOutputStream](#)
7. [Java PrintStream Class](#)

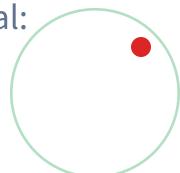
Java Reader/Writer

1. [Java File Class](#)
2. [Java Reader and Writer Class](#)
3. [InputStreamReader and OutputStreamWriter Class](#)
4. [FileReader and FileWriter Class](#)
5. [BufferedReader and BufferedWriter Class](#)
6. [StringReader and StringWriter Class](#)
7. [Java PrintWriter Class](#)

Key Features of Java

Java has remained popular for decades for good reason. Here's what makes it special:

- **Platform Independent:** Write once, run anywhere (thanks to the JVM).





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



Introduction to Java

Welcome to the exciting world of Java! If you're new to programming, you might be wondering what Java is, why it's so popular, and how it can help you create amazing things like apps, games, or even websites. Don't worry if you've never written a line of code before—this blog is just for you! Let's dive into the basics of Java in a simple and friendly way.

What is Java?

Java is a high level class based object-oriented programming language that lets you tell a computer what to do. Think of it as a set of instructions you give to your computer, kind of like a recipe for your favorite dish. Java was created in 1995 by a company called Sun Microsystems (now part of Oracle), and it's been a favorite among developers ever since. Why? Because it's versatile, reliable, and works on almost any device—your phone, laptop, or even a smart fridge!

Why Learn Java?

You might be asking, "Why should I start with Java instead of another language?" Great question! Here are a few reasons why Java is perfect for beginners:

1. **It's Everywhere:** Java powers everything from Android apps to banking systems. If you learn Java, you're learning a skill that's in high demand.



2. **Beginner-Friendly:** Java's rules (called syntax) are straightforward, making it easier to understand than some other languages.
3. **“Write Once, Run Anywhere”:** Java works on any device with a Java Virtual Machine (JVM). You write your code once, and it can run on Windows, Mac, Linux—you name it!
4. **Lots of Help:** Java has a huge community of learners and experts, so if you get stuck, there's always someone or something (like this blog!) to guide you.

How Does Java Work?

Let's break it down simply. When you write Java code, you're giving the computer a list of tasks. But computers don't understand English—they speak in 1s and 0s (called machine language). Java acts like a translator:

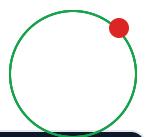
1. You write your code in Java (human-readable).
2. The Java Compiler (`javac`) turns it into something called “bytecode.”
3. The Java Virtual Machine (JVM) takes that bytecode and runs it on your computer, no matter what type it is.

Think of it like this: You're a chef writing a recipe (Java code), the compiler translates it into a universal cooking language (bytecode), and the JVM is the kitchen that cooks it perfectly on any stove!

Your First Java Program

Let's write a super simple program to say “Hello, World!” This is a tradition in programming—it's like your first “Hi” to the coding world.

Here's the code:



```
public class Main {  
    public static void main(String[] args) {  
        System.out.println("Hello, World!");  
    }  
}
```

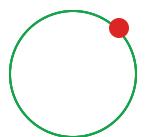


Don't panic if this looks confusing! Let's break it down:

- `public class Main`: This is like naming your recipe “Main.” Every Java program needs a class (a container for your code).
- `public static void main`: This is the starting point of your program—like pressing “Play” on a video. Java looks for this to know where to begin.
- `System.out.println("Hello, World!");`: This tells the computer to print “Hello, World!” on the screen. The semicolon (;) is like a period at the end of a sentence.

To run this:

1. Open any Online Java Compiler or you can use Coding Shuttle’s Online Java Compiler
<https://www.codingshuttle.com/compilers/java/>
2. Copy and Paste the Above code and simply click on Run button, you will see the output Hello World Printed in the Output section.



The screenshot shows the CodingShuttle Online Java Compiler interface. On the left, there's a sidebar with icons for Java, C, C++, Python, Go, and JavaScript. The main area has tabs for "Java Online Compiler", "Share", "Settings", and "Run". A "New" badge is visible above the tabs. A banner at the top says "Limited Time Offer" and "Maximum Discount on All Courses, Use the Coupon Code PAYDAY". The code editor contains a simple "Hello, World!" program. The output panel shows the status as "Accepted", memory usage as 9896 byte, and execution time as 0.083 sec. The output text is "Hello, World!". Below the output is an "Input" field with placeholder text "Enter your input here...". A green speech bubble icon is in the bottom right corner.

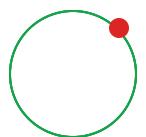
CodingShuttle Online Java Compiler

Tips for Learning Java

- **Start Small:** Don't try to build a game on day one. Begin with simple programs like the one above.
- **Practice:** Write code every day, even if it's just a few lines.
- **Ask Questions:** If something doesn't make sense, search online or ask a friend. Mistakes are part of learning!
- **Have Fun:** Programming is like solving puzzles—enjoy the process.

What's Next?

Now that you've got a taste of Java, you can explore more! Try changing “Hello, World!” to your name or adding two numbers together. In future sections of your Java Handbook, we'll cover topics like loops, conditions, and all the important topics that you must know.





 Pay Day Sale is Now Live!  Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



Download and Install JDK on your Computer

In this blog I will guide you to setup java on your computer so that you can run your first java program using Notepad.

What is JDK and Why it is required?

For Developing Java Program we require JDK (Java Development Kit), It will have compiler and the Runtime environment which will help us to write a java program and execute them.

Once you install JDK along with that you will get two more things that is JRE (Java Runtime Environment) and JVM (Java Virtual Machine).





JDK, JRE and JVM

Let's understand each of this one by one

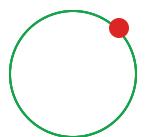
1. JDK - JDK is Java Development Kit it will have all the development tools required for compiling, debugging the java program, and various other tools are available.
2. JRE - Once the program is compiled you have to execute it, for that we require JRE. JRE contains the library of the Java Classes and JVM
3. JVM - JVM comes with JRE and it is the one which actually executes our program.

How all this work together?

Let's say we have a Java file with name **First.java**, when we compile this file with **javac First.** **java** we will get **First.class** a compiled file.

Now to execute **First.class** file we have to simply run **java First** command, it will start executing the file.

So, for compilation we use **javac** and for execution we use **java**.



Download and Install JDK

After understanding JDK, JRE and JVM it's time to download and install JDK on your computer and compile and execute your first java program.

1. Visit :- <https://www.oracle.com/java/technologies/downloads/#jdk24-windows> and install the latest version of jdk, I will install JDK 24 x64 MSI Installer you can choose other options also.

The screenshot shows the Oracle Java Downloads page. At the top, there is a navigation bar with links for Apps, Google, Content Kanban Board, aws home, Super Set, Lucid App, Excalidraw-thumbn..., Content Manager, Meet - baw-zezd-ufy, Our Blogs | Coding..., and All Bookmarks. Below the navigation bar, the Oracle logo is visible along with links for Products, Industries, Resources, Customers, Partners, Developers, and Company. A search bar and a language selection dropdown (American English) are also present. The main content area is titled "Java SE Development Kit 24 downloads". It states that JDK 24 is the latest Long-Term Support (LTS) release of the Java SE Platform. Below this, it says "Earlier JDK versions are available below." A horizontal menu bar at the bottom of this section includes links for JDK 24, JDK 23, JDK 21, GraalVM for JDK 24, GraalVM for JDK 23, and GraalVM for JDK 21. The "Java 24" link is highlighted. The "Windows" tab is selected under the "Java SE Development Kit 24 downloads" section. A table lists three download options:

Product/file description	File size	Download
x64 Compressed Archive	229.40 MB	https://download.oracle.com/java/24/latest/jdk-24_windows-x64_bin.zip (sha256)
x64 Installer	205.76 MB	https://download.oracle.com/java/24/latest/jdk-24_windows-x64_bin.exe (sha256)
x64 MSI Installer	204.51 MB	https://download.oracle.com/java/24/latest/jdk-24_windows-x64_bin.msi (sha256)

Oracle Website

1. Once downloaded simply click on installer file (.exe or .msi or .dmg) and install the setup
2. After installation completes open terminal and verify the java version by running `java --version` command, you will see the java version which you have installed.

The screenshot shows a terminal window with a dark background. The command `java --version` is entered and its output is displayed:
C:\\\\Users\\\\Prem Mane>java --version
java 24 2025-03-18
Java(TM) SE Runtime Environment (build 24+36-3646)
Java HotSpot(TM) 64-Bit Server VM (build 24+36-3646, mixed mode, sharing)
C:\\\\Users\\\\Prem Mane>

Compile and Execute your first java program

Now you have successfully setup a Java Environment on your machine. It's time to write your first hello world program and run it.

1. Create a new file and paste the following code in the file and save the file with name First and extension .java (First.java).

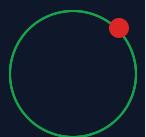
```
public class First {  
    public static void main(String[] args) {  
        System.out.println("Hello, World!");  
    }  
}
```



1. After that open command prompt and go to the the location where you have saved the file.
2. Run this command `javac [First.java](<http://First.java>)` to compile the file, this will create First.class (Compiled file).
3. Now we have to simply execute this compiled file and to execute run this command `java First`

Output:

```
C:\\\\Users\\\\Prem Mane\\\\OneDrive\\\\Desktop\\\\Java Programs>dir  
Volume in drive C is OS  
Volume Serial Number is 42A8-7BC9  
  
Directory of C:\\\\Users\\\\Prem Mane\\\\OneDrive\\\\Desktop\\\\Java Programs  
  
01-04-2025  15:09      <DIR>          .  
01-04-2025  15:08      <DIR>          ..  
01-04-2025  15:11                  122 First.java  
                           1 File(s)           122 bytes
```





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Java Comments

Imagine you've written a Java program that you need to share with your project team. Without comments, your teammates might struggle to understand what your code does. To make your code more understandable, you should add comments explaining its functionality.

Comments are of two types

1. Single-line Comment

In Java, a single-line comment starts and ends in the same line. To write a single-line comment, we can use the `//` symbol. For example,

```
class HelloWorld {  
    public static void main(String[] args) {  
        // print Hello World to the screen  
        System.out.println("Hello World");  
    }  
}
```

2. Multi-line Comment

When we want to write comments in multiple lines, we can use the multi-line comment. To write multi-line comments, we can use the `/*....*/` symbol. For example,



Java Tutorial – A Comprehensive Guide for Beginners

Introduction

Java is one of the most widely used and versatile programming languages in the world. From desktop applications to web services and mobile apps to enterprise systems, Java powers countless technologies and platforms.

This Java tutorial is designed for beginners who want to build a strong foundation in Java programming. Whether you're completely new to programming or transitioning from another language, this guide covers everything from core concepts to essential object-oriented principles and beyond.

If you're aiming to become a proficient Java developer, follow this structured path to master Java, step by step.

Prerequisites

Before you dive into Java, it's helpful (but not mandatory) to be familiar with:

- **Basic Computer Knowledge:** Understanding how software runs, what an IDE is, etc.
- **Logic Building:** Basic problem-solving and logical thinking.



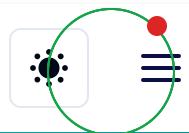
- **Understanding of Programming Concepts** (optional): Knowing variables, loops, or functions from another language is a bonus.

Master These Topics

Once you're ready, follow this structured list to learn Java from the ground up:

1. Introduction to Java
2. Java Fundamentals
3. Java Flow Control
4. Java Arrays
5. Java Methods
6. Java Object-Oriented Programming
7. Java Inner Classes
8. Static and Final Keywords
9. Java Exception Handling
10. Java Generics
11. Java Lambda Expressions
12. Java Collection Framework
13. Java Input/Output Streams
14. Java Reader/Writer

Introduction to Java





1. [Java Variables and Literals](#)
2. [Data Types in Java](#)
3. [Operators in Java](#)
4. [Java Basic Input and Output](#)
5. [Java Expressions, Statements and Blocks](#)

Java Flow Control

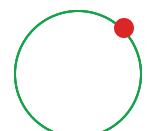
1. [Java if else statement](#)
2. [Java Ternary Operator](#)
3. [Java For Loop](#)
4. [Java while and do while loop](#)
5. [Java continue and break statement](#)
6. [Java Switch statement](#)

Java Arrays

1. [Java Arrays](#)
2. [Java Multidimensional Arrays](#)
3. [Java Copy Arrays](#)

Java Methods

1. [Methods in Java](#)



2. Parameter Passing in Java
3. Method Overloading
4. Variable Arguments
5. Java CommandLine Arguments
6. Java Recursive Method

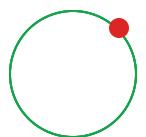
Java Object-Oriented Programming

1. Principles of Object-Oriented Programming
2. Java Class and Objects
3. Java Constructor
4. Encapsulation and Data Hiding
5. Inheritance in Java
6. Constructors in Inheritance
7. this Vs super Keyword
8. Method Overriding
9. Dynamic Method Dispatch
10. Polymorphism using Overloading and Overriding
11. Abstract Classes
12. Interfaces in Java

Java Inner Classes

1. Inner Classes in Java

Static and Final Keywords



1. Static Members and Static Blocks in Java
2. Final Members
3. Singleton Class

Java Exception Handling

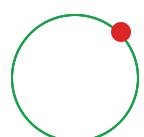
1. What are Exceptions
2. How to handle Exception
3. Try and Catch Block
4. Multiple and Nested Try Catch
5. Class Exception
6. Checked and Unchecked Exceptions
7. Throw Vs Throws
8. Finally Block
9. Try with Resources

Java Generics

1. Introduction to Generics
2. Defining Generic Class
3. Bounds on Generics
4. Java Generic Methods

Java Lambda Expressions

1. Introduction to Lambda Expressions
2. Parameters in Lambda Expressions



3. Capture in Lambda Expression

4. Method References

Java Collection Framework

1. Collections in Java

2. Understanding Collection Interface

3. Understanding List Interface in Java

4. Java ArrayList

5. Java LinkedList

6. Java Vector

7. Java Stack

8. Understanding Queue Interface in Java

9. Java PriorityQueue

10. Java ArrayDeque

11. Understanding Set Interface in Java

12. Java HashSet

13. Java LinkedHashSet

14. Java TreeSet

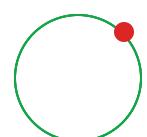
15. Understanding Map Interface in Java

16. Java HashMap

17. Java LinkedHashMap

18. Java TreeMap

19. Comparable Interface in Java



20. Comparator Interface in Java

Java Input/Output Streams

1. [What are Streams](#)
2. [InputStream and OutputStream Classes](#)
3. [FileInputStream and FileOutputStream](#)
4. [ByteArrayInputStream and ByteArrayOutputStream](#)
5. [ObjectInputStream and ObjectOutputStream](#)
6. [BufferedInputStream and BufferedOutputStream](#)
7. [Java PrintStream Class](#)

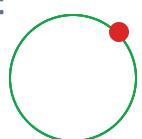
Java Reader/Writer

1. [Java File Class](#)
2. [Java Reader and Writer Class](#)
3. [InputStreamReader and OutputStreamWriter Class](#)
4. [FileReader and FileWriter Class](#)
5. [BufferedReader and BufferedWriter Class](#)
6. [StringReader and StringWriter Class](#)
7. [Java PrintWriter Class](#)

Key Features of Java

Java has remained popular for decades for good reason. Here's what makes it special:

- **Platform Independent:** Write once, run anywhere (thanks to the JVM).





 Pay Day Sale is Now Live!  Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



Java Variables and Literals

In Java, variables and literals are foundational concepts that every beginner must understand. These concepts allow you to store and manipulate data in your programs. This blog will explain what variables and literals are, how to use them, and provide simple code examples to help you grasp these concepts quickly.

What are Variables in Java?

In Java, a **variable** is a container used to store data values. Each variable has a data type that defines what kind of data it can hold (such as integers, floating-point numbers, strings, etc.). The value of a variable can change during the execution of a program.

Syntax for Declaring a Variable

To declare a variable in Java, you need to specify the following:

1. **Data type:** Specifies the type of data the variable will store (e.g., `int`, `String`).
2. **Variable name:** The name of the variable that will be used to reference it.
3. **Value (optional):** You can assign an initial value to the variable during declaration.

Example 1: Declaring and Initializing a Variable



```
class Main {  
    public static void main(String[] args) {  
        // Declare and initialize a variable  
        int age = 25; // 'int' is the data type, 'age' is the variable name, 25 is the value  
  
        // Print the value of the variable  
        System.out.println("Age: " + age); // Output: Age: 25  
    }  
}
```

Explanation:

- The variable `age` is declared with the data type `int` (for integers) and initialized with the value `25`.
- `System.out.println()` prints the value of `age` to the console.

Expected Output:

```
Age: 25
```

What are Literals in Java?

A **literal** is a fixed value that is directly written into the code. Literals are used to assign values to variables. In Java, there are several types of literals, including integer literals, floating-point literals, character literals, string literals, and boolean literals.

Types of Literals

1. **Integer Literals:** These represent integer values (whole numbers).
2. **Floating-point Literals:** These represent numbers with decimal points.
3. **Character Literals:** These represent a single character enclosed in single quotes ('').

4. **String Literals:** These represent a sequence of characters enclosed in double quotes ("").

5. **Boolean Literals:** These represent **true** or **false**.

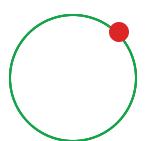
Example 2: Different Types of Literals

```
class Main {  
    public static void main(String[] args) {  
        // Integer literal  
        int num = 100; // '100' is an integer literal  
  
        // Floating-point literal  
        double price = 10.99; // '10.99' is a floating-point literal  
  
        // Character literal  
        char grade = 'A'; // 'A' is a character literal  
  
        // String literal  
        String name = "Alice"; // "Alice" is a string literal  
  
        // Boolean literal  
        boolean isActive = true; // 'true' is a boolean literal  
  
        // Printing the values of all variables  
        System.out.println("Number: " + num); // Output: Number: 100  
        System.out.println("Price: " + price); // Output: Price: 10.99  
        System.out.println("Grade: " + grade); // Output: Grade: A  
        System.out.println("Name: " + name); // Output: Name: Alice  
        System.out.println("Active: " + isActive); // Output: Active: true  
    }  
}
```

Explanation:

- We have used different types of literals to assign values to variables:

- **100** is an **integer literal**.



- `10.99` is a floating-point literal.
 - `'A'` is a character literal.
 - `"Alice"` is a string literal.
 - `true` is a boolean literal.
- Each variable is printed using `System.out.println()`.

Expected Output:

```
Number: 100
Price: 10.99
Grade: A
Name: Alice
Active: true
```



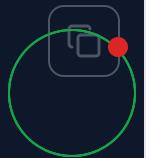
Rules for Naming Variables

When naming variables in Java, there are some rules and conventions to follow:

1. Variable names must start with a letter, dollar sign (\$), or underscore (_).
2. Variable names can contain letters, digits, dollar signs, or underscores, but cannot start with a digit.
3. Java is case-sensitive, so `age` and `Age` would be treated as two different variables.
4. Avoid using reserved keywords (like `int`, `class`, `public`, etc.) as variable names.

Example 3: Valid and Invalid Variable Names

```
class Main {
    public static void main(String[] args) {
        // Valid variable names
```



```
int age = 30;
String firstName = "John";
double $salary = 50000.00;

// Invalid variable names (Uncommenting below lines will cause errors)
// int 1stPlace = 1; // Cannot start with a digit
// double class = 12.5; // 'class' is a reserved keyword

System.out.println("Age: " + age);
System.out.println("Name: " + firstName);
System.out.println("Salary: " + $salary);
}
```

Explanation:

- We have used valid variable names: `age`, `firstName`, and `$salary`.
- The variable `1stPlace` starts with a digit and would result in a compile-time error.
- `class` is a reserved keyword in Java, and thus cannot be used as a variable name.

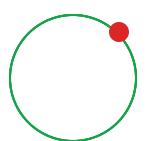
Expected Output:

```
Age: 30
Name: John
Salary: 50000.0
```

Conclusion

In this blog, we've learned the following:

- **Variables** are used to store data in Java programs, and we can assign values to them based on their data types.
- **Literals** are fixed values assigned directly to variables in Java.





 Pay Day Sale is Now Live!  Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



 Java Programming Handbook



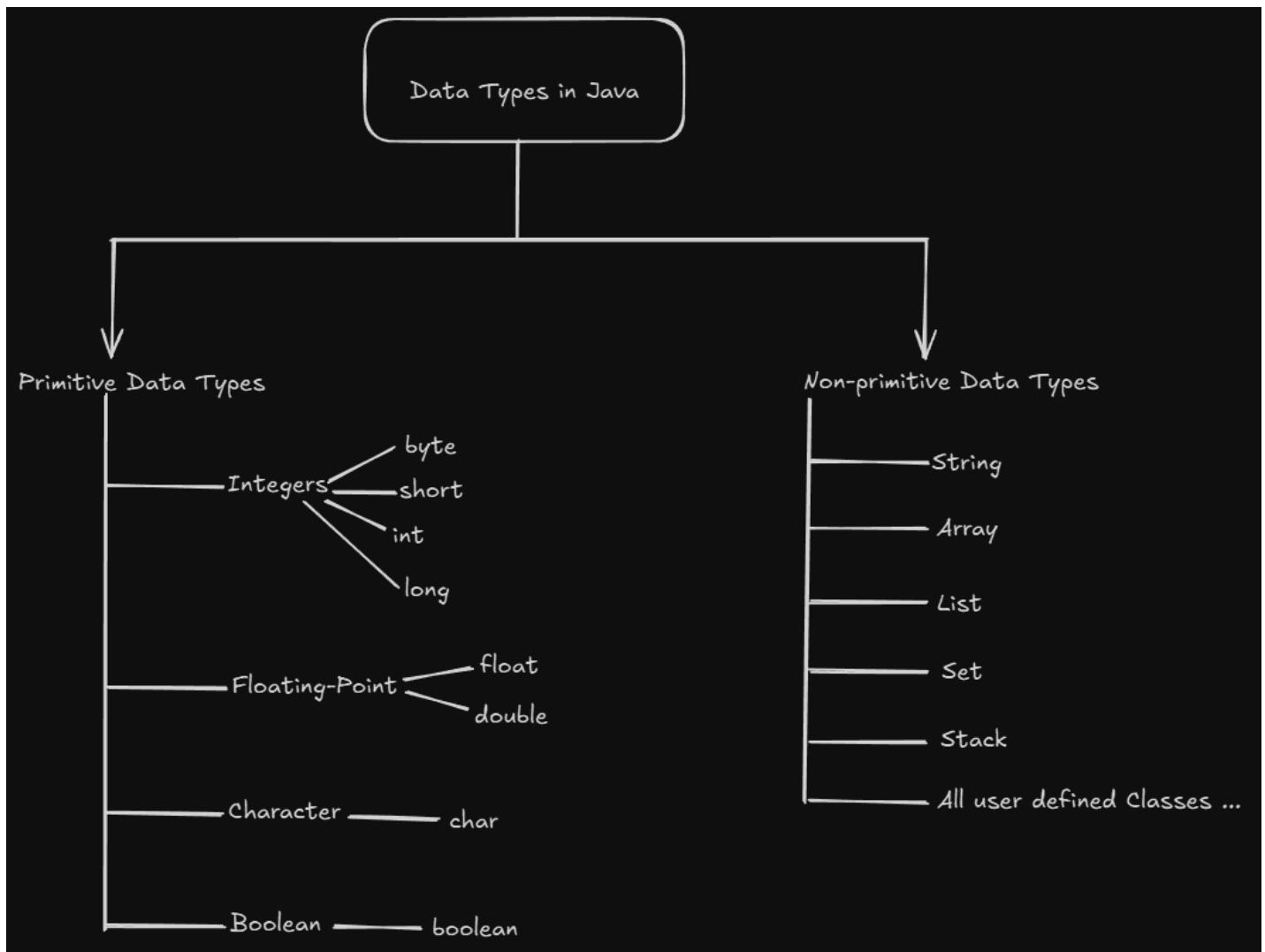
Data Types in Java

When writing Java programs, we need to store and manipulate different types of data, such as numbers, characters, and boolean values. Java provides various data types to define the kind of data a variable can hold. Understanding data types is essential for writing efficient and error-free code.

Types of Data Types in Java

Java data types are broadly categorized into two types:





Data Types in Java

1. Primitive Data Types

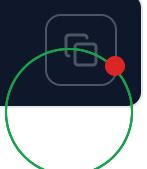
Primitive data types are the most basic types of data. Java has 8 primitive data types:

a) Integer Types

These data types are used to store whole numbers (both positive and negative).

byte: Stores small integers (-128 to 127).

```
byte age = 25;
```



short: Stores numbers from -32,768 to 32,767.

```
short year = 2024;
```



int: Stores larger integers (-2 billion to 2 billion).

```
int salary = 50000;
```



long: Stores very large numbers. Needs an **L** at the end.

```
long population = 7800000000L;
```



b) Floating-Point Types

Used to store decimal numbers.

float: Stores decimal values with less precision. Needs an **F** at the end.

```
float price = 10.99F;
```



double: Stores decimal values with higher precision.

```
double pi = 3.14159265359;
```



c) Character Type

char: Stores a single character using single quotes (' ').

```
char grade = 'A';
```





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Operators in Java

Operators in Java are special symbols that perform operations on variables and values. They help in performing calculations, making decisions, and manipulating data efficiently. Understanding Java operators is essential for writing logical and optimized code.

Types of Operators in Java

Java provides different types of operators, categorized as follows:

1. Arithmetic Operators

Arithmetic operators are used for performing mathematical calculations such as addition, subtraction, multiplication, and division.

```
class Main {  
    public static void main(String[] args) {  
        // Declare variables  
        int num1 = 15, num2 = 4;  
  
        // Performing arithmetic operations  
        System.out.println("num1 + num2 = " + (num1 + num2)); // Output: 19  
        System.out.println("num1 - num2 = " + (num1 - num2)); // Output: 11  
        System.out.println("num1 * num2 = " + (num1 * num2)); // Output: 60  
        System.out.println("num1 / num2 = " + (num1 / num2)); // Output: 3  
        System.out.println("num1 % num2 = " + (num1 % num2)); // Output: 3
```



```
}
```

2. Relational (Comparison) Operators

Relational operators compare two values and return a boolean result (`true` or `false`).

```
class Main {  
    public static void main(String[] args) {  
        // Declare variables  
        int a = 10, b = 20;  
  
        // Performing relational operations  
        System.out.println("a == b: " + (a == b)); // Output: false  
        System.out.println("a != b: " + (a != b)); // Output: true  
        System.out.println("a > b: " + (a > b)); // Output: false  
        System.out.println("a < b: " + (a < b)); // Output: true  
        System.out.println("a >= b: " + (a >= b)); // Output: false  
        System.out.println("a <= b: " + (a <= b)); // Output: true  
    }  
}
```



3. Logical Operators

Logical operators are used to combine multiple conditions.

```
class Main {  
    public static void main(String[] args) {  
        // Declare boolean variables  
        boolean cond1 = true, cond2 = false;  
  
        // Performing Logical operations  
        System.out.println("cond1 && cond2: " + (cond1 && cond2)); // Output: fal  
        System.out.println("cond1 || cond2: " + (cond1 || cond2)); // Output: tru  
        System.out.println("!cond1: " + (!cond1)); // Output: false  
    }  
}
```



4. Bitwise Operators

Bitwise operators perform operations on binary representations of numbers.

```
class Main {  
    public static void main(String[] args) {  
        // Declare variables  
        int x = 5, y = 3;  
  
        // Performing bitwise operations  
        System.out.println("x & y: " + (x & y)); // Output: 1  
        System.out.println("x | y: " + (x | y)); // Output: 7  
        System.out.println("x ^ y: " + (x ^ y)); // Output: 6  
        System.out.println("~x: " + (~x)); // Output: -6  
        System.out.println("x << 1: " + (x << 1)); // Output: 10  
        System.out.println("x >> 1: " + (x >> 1)); // Output: 2  
    }  
}
```



5. Assignment Operators

Assignment operators assign values to variables.

```
class Main {  
    public static void main(String[] args) {  
        // Declare a variable  
        int num = 10;  
  
        // Performing assignment operations  
        num += 5; // Equivalent to num = num + 5  
        System.out.println("num += 5: " + num); // Output: 15  
        num -= 2;  
        System.out.println("num -= 2: " + num); // Output: 13  
        num *= 3;  
        System.out.println("num *= 3: " + num); // Output: 39  
        num /= 2;  
        System.out.println("num /= 2: " + num); // Output: 19  
        num %= 4;  
    }  
}
```



```
        System.out.println("num %= 4: " + num); // Output: 3
    }
}
```

6. Unary Operators

Unary operators work on a single operand.

```
class Main {
    public static void main(String[] args) {
        // Declare a variable
        int a = 5;

        // Performing unary operations
        System.out.println("+a: " + (+a)); // Output: 5
        System.out.println("-a: " + (-a)); // Output: -5
        System.out.println("a++: " + (a++)); // Output: 5 (post-increment)
        System.out.println("After a++: " + a); // Output: 6
        System.out.println("a--: " + (a--)); // Output: 6 (post-decrement)
        System.out.println("After a--: " + a); // Output: 5
    }
}
```

7. Ternary Operator

The ternary operator is a shorthand for **if-else** statements.





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Java Basic Input and Output

In Java, Input and Output (I/O) operations allow us to interact with the outside world, whether it's reading from or writing to a file, console, or even a network. Understanding the basics of I/O in Java is essential for any beginner programmer, as it forms the foundation for user interaction and data management in Java applications.

Types of I/O in Java

Java provides two main types of I/O:

- **Byte-oriented I/O** (for handling binary data, such as images or video files)
- **Character-oriented I/O** (for handling text data)

In this blog, we'll focus on the basics of **console-based I/O** using the **Scanner** class for input and **System.out.println()** for output.

1. Basic Output with **System.out.println()**

The simplest way to print output to the console is using the **System.out.println()** method. This method prints a message followed by a newline.

Example 1: Printing to the Console



```
class Main {  
    public static void main(String[] args) {  
        // Printing a message to the console  
        System.out.println("Hello, World!"); // Output: Hello, World!  
    }  
}
```

Explanation:

- `System.out.println()` prints the string "Hello, World!" to the console.
- `println()` means "print line", so it automatically moves to the next line after printing.

Expected Output:

```
Hello, World!
```

2. Reading Input from the Console using Scanner

To read input from the user, we use the `Scanner` class. This class is part of the `java.util` package and allows us to capture different types of input, such as strings, integers, and floating-point numbers.

Example 2: Reading User Input

```
import java.util.Scanner;  
  
class Main {  
    public static void main(String[] args) {  
        // Create a Scanner object to read input from the console  
        Scanner scanner = new Scanner(System.in);  
  
        // Ask the user for their name  
        System.out.print("Enter your name: ");
```



```
String name = scanner.nextLine(); // Read the entire line of input

// Output the name entered by the user
System.out.println("Hello, " + name + "!");

// Close the scanner
scanner.close();
}

}
```

Explanation:

- We import the `Scanner` class from `java.util`.
- We create a `Scanner` object, `scanner`, to read input from `System.in` (the standard input stream, i.e., the keyboard).
- The `nextLine()` method reads an entire line of text input by the user.
- After reading the input, we print a greeting using `System.out.println()`.

Expected Output:

```
Enter your name: John
Hello, John!
```

3. Reading Different Types of Data

The `Scanner` class provides methods for reading different types of data, such as integers, floating-point numbers, and booleans.

Example 3: Reading Integer Input

```
import java.util.Scanner;
```

```
class Main {  
    public static void main(String[] args) {  
        // Create a Scanner object  
        Scanner scanner = new Scanner(System.in);  
  
        // Ask the user to enter an integer  
        System.out.print("Enter an integer: ");  
        int num = scanner.nextInt(); // Reads the next integer  
  
        // Output the number entered by the user  
        System.out.println("You entered: " + num);  
  
        // Close the scanner  
        scanner.close();  
    }  
}
```

Explanation:

- We use `nextInt()` to read an integer input from the user.
- `System.out.println()` is used to display the entered value.

Expected Output:

```
Enter an integer: 42  
You entered: 42
```

Example 4: Reading a Floating-Point Number

```
import java.util.Scanner;  
  
class Main {  
    public static void main(String[] args) {  
        // Create a Scanner object  
        Scanner scanner = new Scanner(System.in);
```

```
// Ask the user to enter a floating-point number
System.out.print("Enter a floating-point number: ");
double num = scanner.nextDouble(); // Reads the next double

// Output the number entered by the user
System.out.println("You entered: " + num);

// Close the scanner
scanner.close();
}
```

Explanation:

- We use `nextDouble()` to read a floating-point number from the user.
- `System.out.println()` displays the entered number.

Expected Output:

```
Enter a floating-point number: 3.14
```



```
You entered: 3.14
```

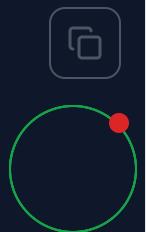
4. Reading Multiple Inputs

You can read multiple values in a single line using the `Scanner` class. Below is an example that reads an integer and a string in one line.

Example 5: Reading Multiple Inputs

```
import java.util.Scanner;

class Main {
    public static void main(String[] args) {
        // Create a Scanner object
```



```
Scanner scanner = new Scanner(System.in);

// Ask the user to enter an integer and a string
System.out.print("Enter an integer and a name: ");

// Read the integer and the string
int age = scanner.nextInt(); // Read the integer
scanner.nextLine(); // Consume the leftover newline character
String name = scanner.nextLine(); // Read the string

// Output the inputs
System.out.println("Name: " + name + ", Age: " + age);

// Close the scanner
scanner.close();
}
```

Explanation:

- After reading the integer using `nextInt()`, we need to call `scanner.nextLine()` to consume the newline character left behind. This allows us to read the string correctly.
- The input values are then printed using `System.out.println()`.

Expected Output:

```
Enter an integer and a name: 25
John
Name: John, Age: 25
```

5. Formatting Output

Java also provides the `printf()` method for formatting output. It allows you to control the output format, such as the number of decimal places or the width of the output.

Example 6: Formatting Output with printf()

```
java
CopyEdit
class Main {
    public static void main(String[] args) {
        // Declare variables
        double price = 23.456;

        // Format output using printf
        System.out.printf("The price is: %.2f\n", price); // Output: The price
    }
}
```

Explanation:

- `%.2f` formats the floating-point number to two decimal places.
- The `printf()` method provides more control over how data is presented.

Expected Output:

```
The price is: 23.46
```

Conclusion

In this blog, we have covered the basics of **Input and Output** in Java. We learned how to use:

- `System.out.println()` to display output.
- `Scanner` class to take input from the user.
- Different methods like `nextLine()`, `nextInt()`, and `nextDouble()` to read various data types.



Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



Java Expressions, Statements, and Blocks

In Java, understanding the concepts of **expressions**, **statements**, and **blocks** is essential for writing well-structured programs. These building blocks of Java help you control the flow of the program and manipulate data. This blog will break down these concepts in simple terms, with code examples that are easy to follow.

What are Java Expressions?

In Java, an **expression** is any valid combination of variables, constants, operators, and method calls that can be evaluated to produce a value. Expressions can range from simple to complex, and they always produce a result.

Types of Expressions

- Arithmetic Expressions:** Involve mathematical operations like addition, subtraction, multiplication, etc.
- Relational Expressions:** Used for comparison (e.g., equals, greater than).
- Logical Expressions:** Involve logical operators like `&&` (AND), `||` (OR).
- Assignment Expressions:** Used to assign values to variables.

Example 1: Arithmetic Expression



```
class Main {  
    public static void main(String[] args) {  
        // Declare variables  
        int a = 5, b = 10;  
  
        // Arithmetic expression  
        int result = a + b * 2; // First multiply, then add (according to operat  
  
        // Print result  
        System.out.println("Result: " + result); // Output: Result: 25  
    }  
}
```

Explanation:

- The expression `a + b * 2` is evaluated first by multiplying `b` (10) by 2, then adding `a` (5) to the result, yielding `25`.
- This demonstrates how expressions produce a value when evaluated.

Expected Output:

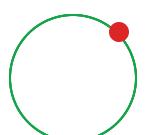
```
Result: 25
```

What are Java Statements?

A **statement** in Java is a complete unit of execution. It is an instruction that the Java compiler can execute. Java programs consist of statements that are executed one after another.

Types of Statements

1. **Declaration Statement:** Used to declare variables.



2. **Expression Statement:** Represents an expression that has side effects (e.g., method calls, assignments).
3. **Control Flow Statements:** Includes conditional statements like `if`, `switch`, and loops like `for`, `while`.
4. **Return Statement:** Exits from a method and optionally returns a value.

Example 2: Expression Statement

```
class Main {  
    public static void main(String[] args) {  
        // Declare a variable  
        int x = 10;  
  
        // Expression statement (assignment)  
        x = x + 5; // x now holds the value 15  
  
        // Print the updated value  
        System.out.println("Updated x: " + x); // Output: Updated x: 15  
    }  
}
```

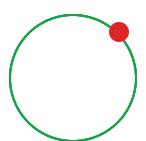
Explanation:

- The statement `x = x + 5;` is an expression statement where the expression `x + 5` is evaluated and assigned to the variable `x`.

Expected Output:

```
Updated x: 15
```

What are Java Blocks?



A **block** in Java is a group of statements enclosed in curly braces `{ }`. Blocks are used to group code together so that it can be executed as a unit. Blocks are often used in methods, loops, conditionals, and class definitions.

Types of Blocks

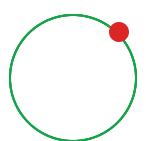
1. **Method Block:** A block inside a method.
2. **Loop Block:** A block inside loops like `for`, `while`.
3. **Conditional Block:** A block inside `if`, `else`, `switch`.
4. **Class Block:** A block that contains class members (fields, methods).

Example 3: Block Inside a Method

```
class Main {  
    public static void main(String[] args) {  
        // Call method to calculate sum  
        int sum = calculateSum(5, 10);  
        System.out.println("Sum: " + sum); // Output: Sum: 15  
    }  
  
    // Method with a block of statements  
    public static int calculateSum(int a, int b) {  
        int result = a + b; // Add a and b  
        return result; // Return the result  
    }  
}
```

Explanation:

- The method `calculateSum` has a block of code that performs the addition of `a` and `b`, and then returns the result.
- The curly braces `{ }` define the scope of the method block.



Expected Output:

```
Sum: 15
```



Example 4: Block Inside a Loop

```
class Main {  
    public static void main(String[] args) {  
        // Print numbers from 1 to 5  
        for (int i = 1; i <= 5; i++) {  
            // Block inside Loop  
            System.out.println(i); // Prints i value  
        }  
    }  
}
```



Explanation:

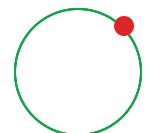
- The block inside the `for` loop contains the statement `System.out.println(i);`, which is executed repeatedly from `i = 1` to `i = 5`.

Expected Output:

```
1  
2  
3  
4  
5
```



Key Differences Between Expressions, Statements, and Blocks



Java Tutorial – A Comprehensive Guide for Beginners

Introduction

Java is one of the most widely used and versatile programming languages in the world. From desktop applications to web services and mobile apps to enterprise systems, Java powers countless technologies and platforms.

This Java tutorial is designed for beginners who want to build a strong foundation in Java programming. Whether you're completely new to programming or transitioning from another language, this guide covers everything from core concepts to essential object-oriented principles and beyond.

If you're aiming to become a proficient Java developer, follow this structured path to master Java, step by step.

Prerequisites

Before you dive into Java, it's helpful (but not mandatory) to be familiar with:

- **Basic Computer Knowledge:** Understanding how software runs, what an IDE is, etc.
- **Logic Building:** Basic problem-solving and logical thinking.



- **Understanding of Programming Concepts** (optional): Knowing variables, loops, or functions from another language is a bonus.

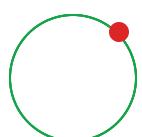
Master These Topics

Once you're ready, follow this structured list to learn Java from the ground up:

1. Introduction to Java
2. Java Fundamentals
3. Java Flow Control
4. Java Arrays
5. Java Methods
6. Java Object-Oriented Programming
7. Java Inner Classes
8. Static and Final Keywords
9. Java Exception Handling
10. Java Generics
11. Java Lambda Expressions
12. Java Collection Framework
13. Java Input/Output Streams
14. Java Reader/Writer

Introduction to Java

1. [Introduction to Java](#)
2. [Installing JDK on your computer](#)



3. Java Comments

Java Fundamentals

1. Java Variables and Literals



Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#).



Java Programming Handbook



Java Flow Control

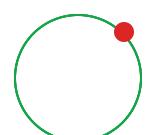
1. [Java if else statement](#)
2. [Java Ternary Operator](#)
3. [Java For Loop](#)
4. [Java while and do while loop](#)
5. [Java continue and break statement](#)
6. [Java Switch statement](#)

Java Arrays

1. [Java Arrays](#)
2. [Java Multidimensional Arrays](#)
3. [Java Copy Arrays](#)

Java Methods

1. [Methods in Java](#)



2. Parameter Passing in Java
3. Method Overloading
4. Variable Arguments
5. Java CommandLine Arguments
6. Java Recursive Method

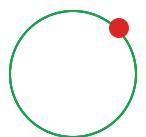
Java Object-Oriented Programming

1. Principles of Object-Oriented Programming
2. Java Class and Objects
3. Java Constructor
4. Encapsulation and Data Hiding
5. Inheritance in Java
6. Constructors in Inheritance
7. this Vs super Keyword
8. Method Overriding
9. Dynamic Method Dispatch
10. Polymorphism using Overloading and Overriding
11. Abstract Classes
12. Interfaces in Java

Java Inner Classes

1. Inner Classes in Java

Static and Final Keywords



1. Static Members and Static Blocks in Java
2. Final Members
3. Singleton Class

Java Exception Handling

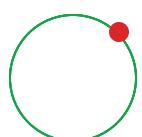
1. What are Exceptions
2. How to handle Exception
3. Try and Catch Block
4. Multiple and Nested Try Catch
5. Class Exception
6. Checked and Unchecked Exceptions
7. Throw Vs Throws
8. Finally Block
9. Try with Resources

Java Generics

1. Introduction to Generics
2. Defining Generic Class
3. Bounds on Generics
4. Java Generic Methods

Java Lambda Expressions

1. Introduction to Lambda Expressions
2. Parameters in Lambda Expressions



3. Capture in Lambda Expression

4. Method References

Java Collection Framework

1. Collections in Java

2. Understanding Collection Interface

3. Understanding List Interface in Java

4. Java ArrayList

5. Java LinkedList

6. Java Vector

7. Java Stack

8. Understanding Queue Interface in Java

9. Java PriorityQueue

10. Java ArrayDeque

11. Understanding Set Interface in Java

12. Java HashSet

13. Java LinkedHashSet

14. Java TreeSet

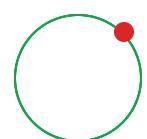
15. Understanding Map Interface in Java

16. Java HashMap

17. Java LinkedHashMap

18. Java TreeMap

19. Comparable Interface in Java



20. Comparator Interface in Java

Java Input/Output Streams

1. [What are Streams](#)
2. [InputStream and OutputStream Classes](#)
3. [FileInputStream and FileOutputStream](#)
4. [ByteArrayInputStream and ByteArrayOutputStream](#)
5. [ObjectInputStream and ObjectOutputStream](#)
6. [BufferedInputStream and BufferedOutputStream](#)
7. [Java PrintStream Class](#)

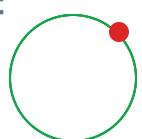
Java Reader/Writer

1. [Java File Class](#)
2. [Java Reader and Writer Class](#)
3. [InputStreamReader and OutputStreamWriter Class](#)
4. [FileReader and FileWriter Class](#)
5. [BufferedReader and BufferedWriter Class](#)
6. [StringReader and StringWriter Class](#)
7. [Java PrintWriter Class](#)

Key Features of Java

Java has remained popular for decades for good reason. Here's what makes it special:

- **Platform Independent:** Write once, run anywhere (thanks to the JVM).





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Java If-Else Statement

The **if-else** statement is one of the most fundamental decision-making structures in programming. It allows your program to execute different code based on whether a condition is true or false. In Java, we use the **if-else** statement to control the flow of execution depending on certain conditions.

In this blog, we'll explore how the **if-else** statement works in Java, provide explanations, and demonstrate examples with expected outputs.

What is an If-Else Statement?

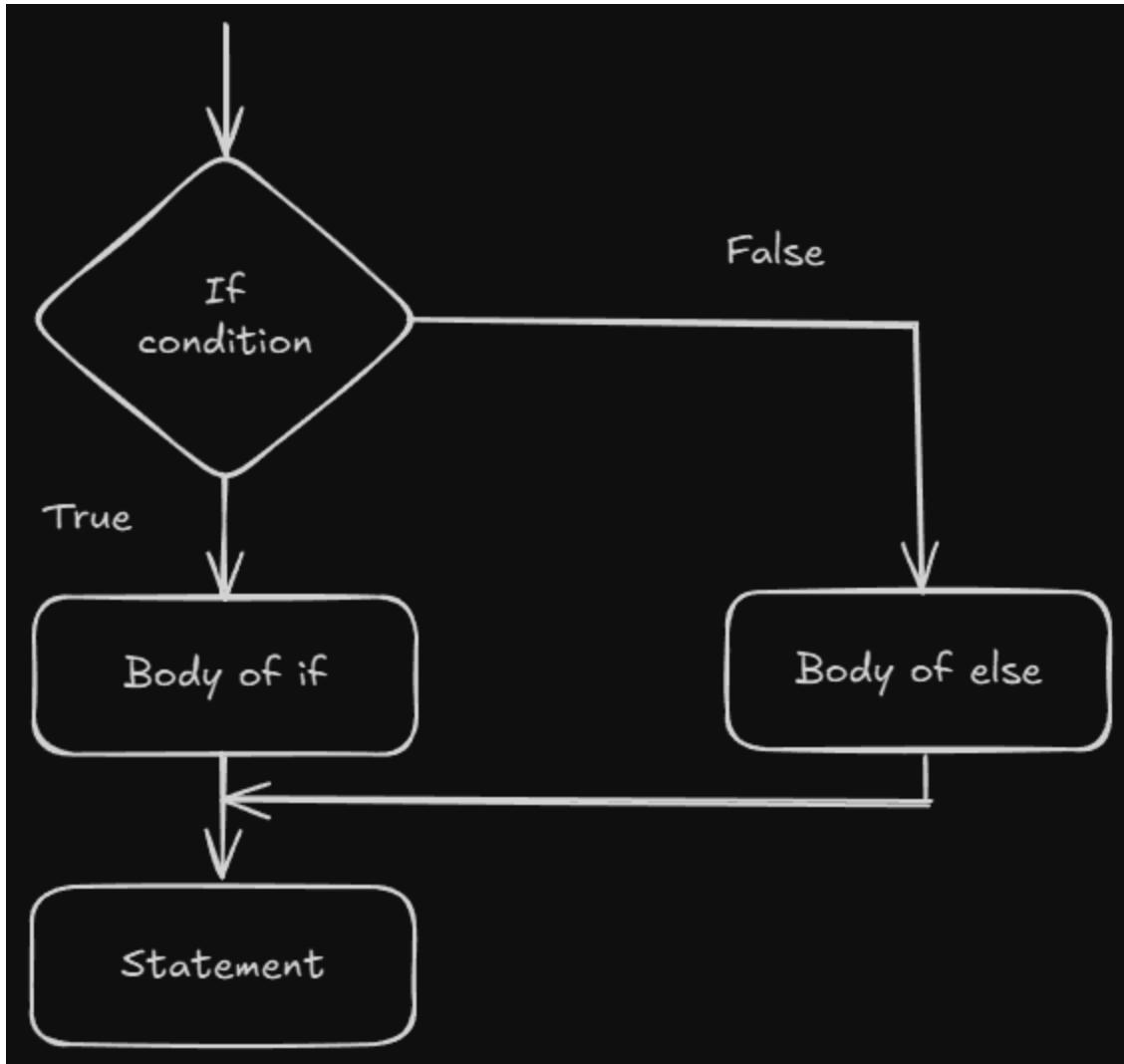
An **if-else** statement is used to execute one block of code if a condition is true, and another block if the condition is false. It's a way to introduce logic in your program, so it can make decisions.

The syntax for an if-else statement in Java is:

```
if (condition) {  
    // code to be executed if the condition is true  
} else {  
    // code to be executed if the condition is false  
}
```



Flow of Execution



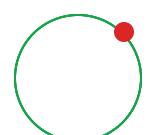
If else flow chart

1. **If condition:** The condition inside the `if` is evaluated. If the condition is `true`, the block of code inside the `if` is executed.
2. **Else block:** If the condition is `false`, the block of code inside the `else` is executed.

Basic Example: If-Else Statement

Let's start with a simple example to see how the `if-else` statement works.

Example 1: Basic If-Else Statement



```
class Main {  
    public static void main(String[] args) {  
        int number = 10;  
  
        // If-Else statement to check if number is positive or negative  
        if (number > 0) {  
            System.out.println("The number is positive.");  
        } else {  
            System.out.println("The number is negative.");  
        }  
    }  
}
```

Explanation:

- The condition `number > 0` is evaluated. Since the value of `number` is 10, which is greater than 0, the condition is `true`, and the code inside the `if` block is executed.
- If the number was negative or zero, the code inside the `else` block would have been executed.

Expected Output:

```
The number is positive.
```

If-Else with Multiple Conditions: Using `else if`

Sometimes, we want to check for more than two conditions. In this case, we can use the `else if` statement to check additional conditions after the initial `if` condition.

Example 2: If-Else-If Statement

```
class Main {  
    public static void main(String[] args) {  
        int number = 0;  
  
        // If-Else-If statement to check if the number is positive, negative, or  
        if (number > 0) {  
            System.out.println("The number is positive.");  
        } else if (number < 0) {  
            System.out.println("The number is negative.");  
        } else {  
            System.out.println("The number is zero.");  
        }  
    }  
}
```

Explanation:

- The first condition checks if the number is positive. Since the number is 0, it moves to the **else if** condition and checks if the number is negative.
- Finally, since the number is neither positive nor negative, the **else** block is executed, and the program prints "The number is zero."

Expected Output:

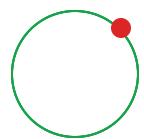
The number is zero.



Nested If-Else Statement

A **nested if-else** statement is when you place an **if-else** statement inside another **if** or **else** block. This is useful when you have more complex conditions to check.

Example 3: Nested If-Else Statement



```
class Main {  
    public static void main(String[] args) {  
        int age = 18;  
  
        // Nested if-else to check voting eligibility  
        if (age >= 18) {  
            if (age == 18) {  
                System.out.println("You are 18 years old and eligible to vote for the  
            } else {  
                System.out.println("You are eligible to vote.");  
            }  
        } else {  
            System.out.println("You are not eligible to vote.");  
        }  
    }  
}
```

Explanation:

- The first **if** checks if the age is greater than or equal to 18.
- Inside the **if** block, there's another **if** that checks if the age is exactly 18.
- If the age is 18, it prints a specific message for first-time voters. Otherwise, it simply prints that the person is eligible to vote.

Expected Output:

```
You are 18 years old and eligible to vote for the first time.
```

If-Else with Logical Operators

You can also combine multiple conditions in an **if** statement using **logical operators** like **&&** (AND), **||** (OR), and **!** (NOT). This helps you to create more complex conditions.

Example 4: If-Else with Logical AND (&&)

```
class Main {  
    public static void main(String[] args) {  
        int age = 25;  
        boolean hasVoterID = true;  
  
        // If-Else with Logical AND (&&) to check voting eligibility  
        if (age >= 18 && hasVoterID) {  
            System.out.println("You are eligible to vote.");  
        } else {  
            System.out.println("You are not eligible to vote.");  
        }  
    }  
}
```

Explanation:

- The condition `age >= 18 && hasVoterID` checks if both conditions are true. Since both are true in this case (age is 25 and the person has a voter ID), the `if` block is executed.

Expected Output:

```
You are eligible to vote.
```

Example 5: If-Else with Logical OR (||)

```
class Main {  
    public static void main(String[] args) {  
        int age = 17;  
        boolean hasParentalConsent = true;  
  
        // If-Else with Logical OR (||) to check voting eligibility with consent  
        if (age >= 18 || hasParentalConsent) {  
            System.out.println("You are eligible to vote.");  
        }  
    }  
}
```



Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Java Ternary Operator

In Java, the **ternary operator** is a compact way of writing an **if-else** statement. It is sometimes referred to as a **conditional operator**. The ternary operator allows you to assign a value based on a condition, making your code more concise and readable.

In this blog, we'll explore the **ternary operator**, how it works, and provide simple examples for better understanding.

What is the Ternary Operator?

The ternary operator in Java is a shorthand version of the **if-else** statement. It works by evaluating a boolean condition and returning one of two values based on whether the condition is **true** or **false**.

Syntax of the Ternary Operator

```
condition ? value_if_true : value_if_false;
```



- **condition:** The boolean expression that is evaluated.
- **value_if_true:** The value returned if the condition is **true**.
- **value_if_false:** The value returned if the condition is **false**.



The ternary operator can be used to assign a value to a variable, or even directly return a result in a method.

Example: Basic Ternary Operator

Let's start with a simple example to understand how the ternary operator works.

Example 1: Basic Usage of the Ternary Operator

```
class Main {  
    public static void main(String[] args) {  
        int number = 5;  
  
        // Ternary operator to check if the number is positive or negative  
        String result = (number > 0) ? "Positive" : "Negative";  
  
        // Print the result  
        System.out.println("The number is: " + result);  
    }  
}
```

Explanation:

- The condition `(number > 0)` checks if the number is greater than zero.
- If the condition is true, it returns `"Positive"`.
- If the condition is false, it returns `"Negative"`.
- In this case, since `number` is 5, which is greater than zero, the condition evaluates to `true`, and the result is `"Positive"`.

Expected Output:



```
The number is: Positive
```

Ternary Operator with Multiple Conditions

You can also use the ternary operator with multiple conditions by nesting it. This is helpful when you want to handle more than two cases.

Example 2: Nested Ternary Operator

```
class Main {  
    public static void main(String[] args) {  
        int number = 0;  
  
        // Nested ternary operator to check if the number is positive, negative,  
        String result = (number > 0) ? "Positive" : (number < 0) ? "Negative" : "  
        // Print the result  
        System.out.println("The number is: " + result);  
    }  
}
```

Explanation:

- First, it checks if the number is positive. If true, it returns **"Positive"**.
- If false, it moves to the second condition and checks if the number is negative. If true, it returns **"Negative"**.
- If both conditions are false, it returns **"Zero"**.

Expected Output:

```
The number is: Zero
```

Ternary Operator for Assignment

The ternary operator can also be used for assigning values to variables in a more compact way, rather than using an `if-else` block.

Example 3: Assigning Values Using Ternary Operator

```
class Main {  
    public static void main(String[] args) {  
        int age = 18;  
  
        // Using ternary operator to assign a value based on the age  
        String eligibility = (age >= 18) ? "Eligible to Vote" : "Not Eligible to  
  
        // Print the eligibility status  
        System.out.println("Eligibility: " + eligibility);  
    }  
}
```

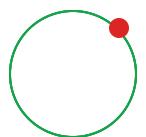
Explanation:

- The condition `(age >= 18)` checks if the age is 18 or older.
- If the condition is true, it assigns `"Eligible to Vote"` to the `eligibility` variable.
- If false, it assigns `"Not Eligible to Vote"`.

Expected Output:

```
Eligibility: Eligible to Vote
```

Ternary Operator for Returning Values



In addition to assigning values, the ternary operator can also be used for returning values from a method.

Example 4: Using Ternary Operator in a Method

```
class Main {  
    // Method that returns a message based on age using ternary operator  
    public static String checkEligibility(int age) {  
        return (age >= 18) ? "Eligible for Adult Activities" : "Not Eligible for  
    }  
  
    public static void main(String[] args) {  
        int age = 20;  
  
        // Call method and print result  
        System.out.println(checkEligibility(age));  
    }  
}
```

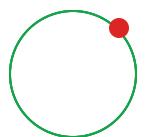
Explanation:

- The ternary operator is used inside the `checkEligibility` method to determine if a person is eligible for adult activities based on their age.
- If the age is 18 or older, it returns `"Eligible for Adult Activities"`. Otherwise, it returns `"Not Eligible for Adult Activities"`.

Expected Output:

```
Eligible for Adult Activities
```

Advantages of Using the Ternary Operator



- **Concise Code:** The ternary operator helps to write compact and concise code, especially when assigning values or making simple decisions.
- **Improves Readability:** For simple conditions, using the ternary operator can improve readability by reducing the lines of code.
- **Faster Decision-Making:** The ternary operator evaluates a condition and returns a result in one line, making it easier to manage simple conditional logic.

When to Avoid the Ternary Operator

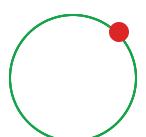
While the ternary operator is convenient, it should be avoided in the following situations:

- **Complex Conditions:** If the condition and outcomes are too complex, using the ternary operator may make the code harder to read. In such cases, it is better to use the traditional `if-else` statement for clarity.
- **Multiple Nested Ternary Operators:** Overusing the ternary operator, especially with multiple nested conditions, can lead to confusing code. It's important to maintain readability.

Conclusion

In this blog, we've learned:

- The **ternary operator** is a shorthand for the `if-else` statement, allowing you to evaluate a condition and return one of two values based on the condition.
- We explored simple and **nested ternary operators** to handle more than two outcomes.
- The **ternary operator** is useful for assigning values and returning results based on conditions, making your code more concise and readable.





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Java For Loop

In Java, a **for loop** is one of the most commonly used control flow statements. It allows you to repeatedly execute a block of code for a fixed number of iterations. This makes it an essential tool for performing repetitive tasks, especially when you know in advance how many times you need to execute a particular statement or block of code.

In this blog, we'll explore how the **for loop** works, its syntax, and provide examples to help you understand its usage clearly.

What is a For Loop?

The **for loop** is used when you know the number of iterations you need to execute. It is ideal for iterating through collections, arrays, or performing repetitive tasks.

Syntax of a For Loop

```
for(initialization; condition; update) {  
    // Code to be executed  
}
```

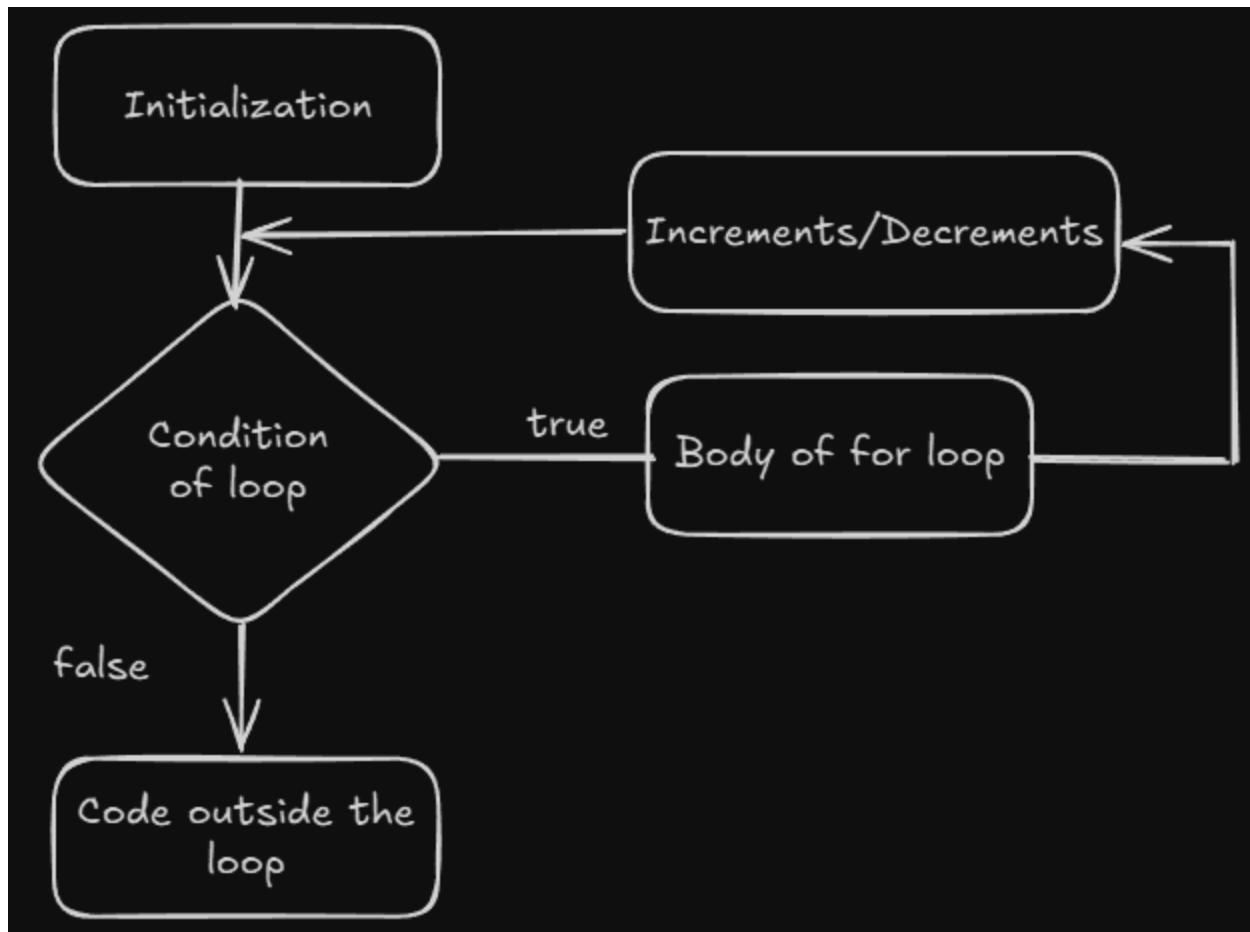


- **initialization:** This step is executed once, before the loop starts. It is typically used to initialize loop control variables.



- **condition:** This is evaluated before each iteration. If the condition evaluates to **true**, the loop body is executed. If it evaluates to **false**, the loop terminates.
- **update:** After each iteration, the update statement is executed. It is typically used to modify the loop control variable (like incrementing or decrementing).

Flow Chart:

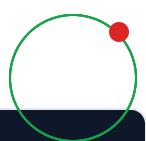


For loop flow chart

Basic Example of a For Loop

Let's start with a simple example to demonstrate the basic structure of a for loop.

Example 1: Print Numbers from 1 to 5



```
class Main {  
    public static void main(String[] args) {  
        // Using a for Loop to print numbers from 1 to 5  
        for (int i = 1; i <= 5; i++) {  
            System.out.println(i);  
        }  
    }  
}
```

Explanation:

- **Initialization:** `int i = 1` sets the starting value of `i` to 1.
- **Condition:** `i <= 5` checks if `i` is less than or equal to 5.
- **Update:** `i++` increments `i` by 1 after each iteration.
- This loop prints the numbers from 1 to 5.

Expected Output:

```
1  
2  
3  
4  
5
```

For Loop with Multiple Statements

You can also have multiple statements in the initialization, condition, or update sections of a for loop. This is useful when you need to modify more than one variable.

Example 2: Multiple Statements in For Loop

```
class Main {  
    public static void main(String[] args) {  
        // Using a for Loop with multiple statements  
        for (int i = 1, j = 10; i <= 5; i++, j -= 2) {  
            System.out.println("i: " + i + ", j: " + j);  
        }  
    }  
}
```



Explanation:

- **Initialization:** `int i = 1, j = 10` initializes two variables, `i` and `j`.
- **Condition:** `i <= 5` checks if `i` is less than or equal to 5.
- **Update:** `i++` increments `i` by 1, and `j -= 2` decrements `j` by 2 after each iteration.
- This loop prints the values of `i` and `j` in each iteration.

Expected Output:

```
i: 1, j: 10  
i: 2, j: 8  
i: 3, j: 6  
i: 4, j: 4  
i: 5, j: 2
```



For Loop with Arrays

A **for loop** is frequently used to iterate through arrays. This is useful when you need to perform an operation on each element in the array.

Example 3: Iterate Through an Array



```
class Main {  
    public static void main(String[] args) {  
        int[] numbers = {2, 4, 6, 8, 10};  
  
        // Using a for Loop to iterate through the array  
        for (int i = 0; i < numbers.length; i++) {  
            System.out.println("Element at index " + i + ": " + numbers[i]);  
        }  
    }  
}
```

Explanation:

- **Initialization:** `int i = 0` starts the loop at the first element of the array.
- **Condition:** `i < numbers.length` ensures the loop continues as long as `i` is less than the length of the array.
- **Update:** `i++` increments `i` by 1 in each iteration.
- The loop prints each element in the `numbers` array.

Expected Output:

```
Element at index 0: 2  
Element at index 1: 4  
Element at index 2: 6  
Element at index 3: 8  
Element at index 4: 10
```

For Loop with a Decrementing Counter

A **for loop** can also be used to count downwards, by using a decrementing counter. This is useful when you need to execute a block of code in reverse order.

Example 4: Count Downwards from 5

```
class Main {  
    public static void main(String[] args) {  
        // Using a for Loop to count down from 5 to 1  
        for (int i = 5; i > 0; i--) {  
            System.out.println(i);  
        }  
    }  
}
```

Explanation:

- **Initialization:** `int i = 5` starts the loop at 5.
- **Condition:** `i > 0` ensures the loop runs while `i` is greater than 0.
- **Update:** `i--` decrements `i` by 1 after each iteration.
- The loop prints numbers from 5 to 1 in reverse order.

Expected Output:

```
5  
4  
3  
2  
1
```

Infinite For Loop

A **for loop** can be set to run infinitely by leaving out the condition, or setting it to always be `true`. This is generally used when you need a loop to run indefinitely until a break condition is met.

Example 5: Infinite For Loop

```
class Main {  
    public static void main(String[] args) {  
        // Infinite for loop  
        for (;;) {  
            System.out.println("This will print forever!");  
            break; // Breaking out of the infinite loop  
        }  
    }  
}
```



Explanation:

- The `for(;;)` creates an infinite loop, since the condition is always `true`.
- The `break` statement immediately exits the loop after the first print statement.
- Without the `break`, the loop would run forever.

Expected Output:

```
This will print forever!
```

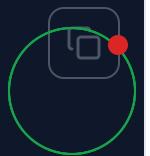


For-Each Loop (Enhanced For Loop)

In addition to the traditional `for loop`, Java also provides a `for-each loop` to simplify the iteration over arrays or collections.

Example 6: For-Each Loop

```
class Main {  
    public static void main(String[] args) {  
        int[] numbers = {1, 2, 3, 4, 5};  
    }  
}
```



```
// Using a for-each Loop to iterate through the array
for (int num : numbers) {
    System.out.println(num);
}
```

Explanation:

- The **for-each loop** simplifies iterating over an array.
- It automatically accesses each element in the array one by one, without needing an index.

Expected Output:

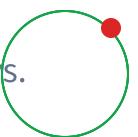
```
1
2
3
4
5
```



Conclusion

In this blog, we learned:

- The **for loop** is a fundamental control flow statement in Java used for iterating over a fixed number of iterations.
- We explored how to use the **for loop** with initialization, conditions, and updates.
- We also learned how to use the **for loop** with arrays and collections, as well as for counting upwards or downwards.
- Additionally, we looked at the **for-each loop** as a simpler way to iterate over arrays.





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Java While and Do-While Loop

In Java, loops are used to execute a block of code repeatedly based on a given condition. While loops and do-while loops are two types of loops in Java that allow you to do this.

In this blog, we'll explore the **while loop** and **do-while loop**, explain their syntax, and provide examples to help you understand when and how to use them.

What is a While Loop?

The **while loop** repeatedly executes a block of code as long as a given condition is **true**. It's often used when you don't know exactly how many times you need to loop, but you know the condition to stop.

Syntax of a While Loop

```
while (condition) {  
    // Code to be executed  
}
```



- **condition:** This is the test condition that's evaluated before each iteration. If the condition evaluates to **true**, the loop body is executed. If the condition evaluates to **false**, the loop terminates.
- The loop keeps running until the condition becomes **false**.



Basic Example of a While Loop

Let's start with a simple example to demonstrate how a **while loop** works.

Example 1: Print Numbers from 1 to 5

```
class Main {  
    public static void main(String[] args) {  
        int i = 1;  
  
        // Using a while loop to print numbers from 1 to 5  
        while (i <= 5) {  
            System.out.println(i);  
            i++; // Incrementing the value of i  
        }  
    }  
}
```

Explanation:

- **Initialization:** `int i = 1` sets the initial value of `i`.
- **Condition:** `i <= 5` ensures the loop runs as long as `i` is less than or equal to 5.
- **Update:** `i++` increments `i` by 1 after each iteration.
- The loop prints the numbers from 1 to 5.

Expected Output:

```
1  
2  
3  
4  
5
```

What is a Do-While Loop?

The **do-while loop** is similar to the while loop, but with one key difference: the condition is checked **after** the loop's body is executed. This guarantees that the loop body will run at least once, even if the condition is false.

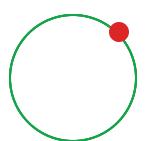
Syntax of a Do-While Loop

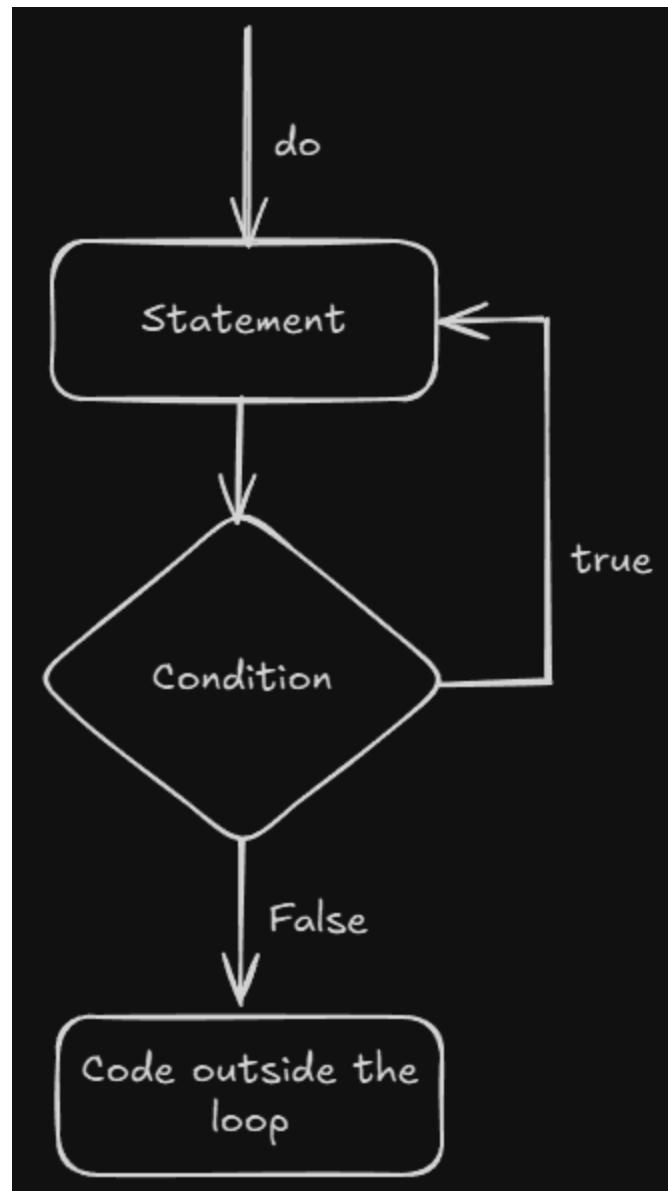
```
do {  
    // Code to be executed  
} while (condition);
```



- The **condition** is checked after the loop executes the code block.
- The loop body executes **at least once**, regardless of the condition.

Flow chart:





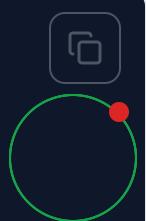
Java do-while loop flow chart

Basic Example of a Do-While Loop

Let's now see a simple example to understand how the do-while loop works.

Example 2: Print Numbers from 1 to 5 using Do-While Loop

```
class Main {  
    public static void main(String[] args) {  
        int i = 1;  
  
        // Using a do-while Loop to print numbers from 1 to 5
```



```
do {
    System.out.println(i);
    i++; // Incrementing the value of i
} while (i <= 5);
}
```

Explanation:

- **Initialization:** `int i = 1` sets the initial value of `i`.
- **Condition:** `i <= 5` ensures the loop runs as long as `i` is less than or equal to 5.
- **Update:** `i++` increments `i` by 1 after each iteration.
- The loop prints the numbers from 1 to 5, but it will always execute the body at least once, even if `i` were initially greater than 5.

Expected Output:

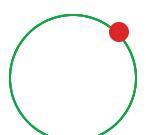
```
1
2
3
4
5
```



Difference Between While and Do-While Loop

The primary difference between the **while loop** and the **do-while loop** is when the condition is checked:

- In the **while loop**, the condition is checked **before** the loop starts executing, which means the code inside the loop may never execute if the condition is initially false.



- In the **do-while loop**, the condition is checked **after** the loop has executed, ensuring that the loop body runs **at least once**, even if the condition is false initially.

Example 3: Do-While Loop Always Runs at Least Once

```
class Main {  
    public static void main(String[] args) {  
        int i = 6; // Initial value greater than 5  
  
        // Using a do-while Loop  
        do {  
            System.out.println("This will print at least once");  
            i++;  
        } while (i <= 5); // Condition is false, but the Loop runs once  
    }  
}
```

Explanation:

- Even though the condition `i <= 5` is false from the start, the loop executes the block of code at least once because it's a **do-while loop**.

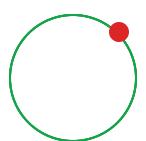
Expected Output:

```
This will print at least once
```

Infinite Loops

Both the **while loop** and **do-while loop** can be used to create an **infinite loop** if the condition is always **true**. This is useful when you need the program to keep running until some external condition occurs (like user input or a system event).

Example 4: Infinite While Loop



```
class Main {  
    public static void main(String[] args) {  
        // Infinite Loop with while  
        while (true) {  
            System.out.println("This loop will run forever!");  
            break; // Breaking out of the infinite loop  
        }  
    }  
}
```



Explanation:

- The condition **true** makes the while loop run indefinitely.
- The **break** statement is used to exit the loop after one iteration.

Expected Output:

```
This loop will run forever!
```

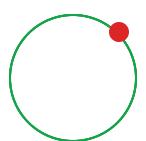


Example 5: Infinite Do-While Loop

```
class Main {  
    public static void main(String[] args) {  
        // Infinite Loop with do-while  
        do {  
            System.out.println("This do-while loop will run forever!");  
            break; // Breaking out of the infinite loop  
        } while (true);  
    }  
}
```



Explanation:





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



Java Continue and Break Statement

In Java, loops are often used to execute a block of code repeatedly. However, sometimes we need to skip certain iterations or stop the loop entirely based on a specific condition. This is where the `continue` and `break` statements come into play. They help control the flow of the loop.

In this blog, we will explore how to use the `continue` and `break` statements, understand their purpose, and see how they work in different types of loops.

What is the `continue` Statement?

The `continue` statement in Java is used to skip the current iteration of a loop and continue with the next iteration. When the `continue` statement is encountered inside a loop, it forces the loop to move directly to the next iteration without executing the remaining code in the current iteration.

Syntax of `continue` Statement

```
continue;
```



The `continue` statement can only be used inside loops (`for`, `while`, `do-while`). When encountered, it skips the remaining code inside the loop for the current iteration and j



the next iteration of the loop.

Example 1: Using continue in a for Loop

Let's see how the `continue` statement works in a `for` loop.

```
class Main {  
    public static void main(String[] args) {  
        // Looping from 1 to 10  
        for (int i = 1; i <= 10; i++) {  
            // Skip even numbers  
            if (i % 2 == 0) {  
                continue; // Skip the rest of the loop body for even numbers  
            }  
            System.out.println(i); // Print odd numbers  
        }  
    }  
}
```

Explanation:

- The `if (i % 2 == 0)` condition checks if the number is even.
- If the number is even, the `continue` statement is triggered, and the `System.out.println(i);` is skipped for that iteration.
- As a result, only odd numbers (1, 3, 5, 7, 9) are printed.

Expected Output:

```
1  
3  
5  
7  
9
```

What is the **break** Statement?

The **break** statement in Java is used to terminate the execution of the loop immediately, regardless of the loop's condition. When the **break** statement is executed inside a loop, it exits the loop and continues with the next statement after the loop.

Syntax of break Statement

```
break;
```



The **break** statement can be used in any loop (**for**, **while**, **do-while**) and in switch statements to exit early.

Example 2: Using **break** in a **for** Loop

Let's see how the **break** statement works in a **for** loop.

```
class Main {  
    public static void main(String[] args) {  
        // Looping from 1 to 10  
        for (int i = 1; i <= 10; i++) {  
            if (i == 6) {  
                break; // Exit the Loop when i equals 6  
            }  
            System.out.println(i); // Print numbers from 1 to 5  
        }  
    }  
}
```



Explanation:

- The loop starts from 1 and prints numbers.
- When **i** equals 6, the **break** statement is triggered, and the loop is immediately exited.



- The loop terminates before printing numbers 6 to 10.

Expected Output:

```
1  
2  
3  
4  
5
```



Using continue and break in a while Loop

Both the `continue` and `break` statements can also be used in `while loops`. Let's see some examples.

Example 3: Using continue in a while Loop

```
class Main {  
    public static void main(String[] args) {  
        int i = 1;  
  
        // Looping from 1 to 10  
        while (i <= 10) {  
            if (i % 2 == 0) {  
                i++; // Increment the value of i before continuing  
                continue; // Skip even numbers  
            }  
            System.out.println(i); // Print odd numbers  
            i++;  
        }  
    }  
}
```



Explanation:



- The **continue** statement skips the printing of even numbers.
- The loop prints only odd numbers from 1 to 9.

Expected Output:

```
1  
3  
5  
7  
9
```



Example 4: Using break in a while Loop

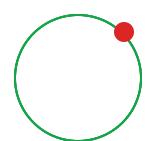
```
class Main {  
    public static void main(String[] args) {  
        int i = 1;  
  
        // Looping from 1 to 10  
        while (i <= 10) {  
            if (i == 6) {  
                break; // Exit the Loop when i equals 6  
            }  
            System.out.println(i); // Print numbers from 1 to 5  
            i++;  
        }  
    }  
}
```



Explanation:

- The loop prints numbers from 1 to 5.
- The **break** statement is triggered when **i** equals 6, and the loop is terminated.

Expected Output:



1
2
3
4
5



continue and break with Nested Loops

In Java, you can also use `continue` and `break` in nested loops (loops inside loops). By default, these statements affect only the innermost loop. However, you can use labeled `continue` and `break` to control outer loops as well.

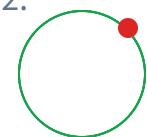
Example 5: Using continue in Nested Loops

```
class Main {  
    public static void main(String[] args) {  
        for (int i = 1; i <= 3; i++) {  
            for (int j = 1; j <= 3; j++) {  
                if (i == 2 && j == 2) {  
                    continue; // Skip the iteration where i == 2 and j == 2  
                }  
                System.out.println("i = " + i + ", j = " + j);  
            }  
        }  
    }  
}
```



Explanation:

- The outer loop iterates over `i`, and the inner loop iterates over `j`.
- The `continue` statement skips the iteration when both `i` equals 2 and `j` equals 2.
- The loop will print all pairs of `i` and `j`, except when `i == 2` and `j == 2`.



Expected Output:

```
i = 1, j = 1
i = 1, j = 2
i = 1, j = 3
i = 2, j = 1
i = 2, j = 3
i = 3, j = 1
i = 3, j = 2
i = 3, j = 3
```



Example 6: Using break in Nested Loops

```
class Main {
    public static void main(String[] args) {
        for (int i = 1; i <= 3; i++) {
            for (int j = 1; j <= 3; j++) {
                if (i == 2 && j == 2) {
                    break; // Exit the inner loop when i == 2 and j == 2
                }
                System.out.println("i = " + i + ", j = " + j);
            }
        }
    }
}
```



Explanation:

- The **break** statement exits the inner loop when **i == 2** and **j == 2**.
- The inner loop terminates early, and the outer loop continues its next iteration.

Expected Output:





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



Java switch Statement

The **switch** statement in Java is used when we need to test a variable against multiple possible values. It is an alternative to using multiple **if-else-if** statements and makes the code more readable and efficient.

In this blog, we will cover:

- What is the **switch** statement?
- How does the **switch** statement work?
- Syntax of **switch**
- Example programs with explanations
- Rules and limitations of **switch**
- Enhanced **switch** statement in Java 12+

What is the **switch** Statement?

A **switch** statement allows a variable to be tested against multiple values, executing different blocks of code depending on which value matches.

Instead of writing multiple **if-else-if** conditions, we can use **switch** for better readability.



How Does the switch Statement Work?

- The `switch` statement evaluates an **expression**.
- The result of the expression is compared with multiple **case values**.
- If a match is found, the corresponding block of code executes.
- The `break` statement stops execution after a case is matched.
- If no cases match, the `default` case (if provided) is executed.

Syntax of switch Statement

```
switch (expression) {  
    case value1:  
        // Code to execute if expression == value1  
        break;  
    case value2:  
        // Code to execute if expression == value2  
        break;  
    default:  
        // Code to execute if no case matches  
}
```

Example 1: Basic switch Statement

```
class Main {  
    public static void main(String[] args) {  
        int day = 3; // 1 - Monday, 2 - Tuesday, ..., 7 - Sunday  
  
        switch (day) {  
            case 1:  
                System.out.println("Monday");  
                break;  
            case 2:
```

```
        System.out.println("Tuesday");
        break;
    case 3:
        System.out.println("Wednesday");
        break;
    case 4:
        System.out.println("Thursday");
        break;
    case 5:
        System.out.println("Friday");
        break;
    case 6:
        System.out.println("Saturday");
        break;
    case 7:
        System.out.println("Sunday");
        break;
    default:
        System.out.println("Invalid day");
    }
}
}
```

Expected Output

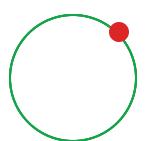
Wednesday



Explanation:

- The variable **day** is **3**, so it matches **case 3:**.
- "**Wednesday**" is printed.
- The **break** statement exits the **switch** block.

Example 2: switch with char



```
class Main {  
    public static void main(String[] args) {  
        char grade = 'B';  
  
        switch (grade) {  
            case 'A':  
                System.out.println("Excellent!");  
                break;  
            case 'B':  
                System.out.println("Good job!");  
                break;  
            case 'C':  
                System.out.println("You passed.");  
                break;  
            case 'D':  
                System.out.println("Better luck next time.");  
                break;  
            default:  
                System.out.println("Invalid grade");  
        }  
    }  
}
```



Expected Output

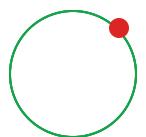
Good job!



Explanation:

- The variable `grade` is `'B'`, so it matches `case 'B':`.
- "Good job!" is printed.

Example 3: switch without break



If we do not use the `break` statement, execution will continue to the next case.

```
class Main {  
    public static void main(String[] args) {  
        int num = 2;  
  
        switch (num) {  
            case 1:  
                System.out.println("One");  
            case 2:  
                System.out.println("Two");  
            case 3:  
                System.out.println("Three");  
            default:  
                System.out.println("Invalid number");  
        }  
    }  
}
```

Expected Output

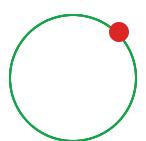
```
Two  
Three  
Invalid number
```

Explanation:

- The value `2` matches `case 2:`.
- Since there's no `break`, execution continues to the next cases.

Example 4: switch with Strings (Java 7+)

Java allows `switch` statements with `String` values starting from Java 7.



```
class Main {  
    public static void main(String[] args) {  
        String fruit = "Apple";  
  
        switch (fruit) {  
            case "Apple":  
                System.out.println("Apples are red or green.");  
                break;  
            case "Banana":  
                System.out.println("Bananas are yellow.");  
                break;  
            case "Orange":  
                System.out.println("Oranges are orange.");  
                break;  
            default:  
                System.out.println("Unknown fruit");  
        }  
    }  
}
```

Expected Output

Apples are red or green.

Example 5: Nested switch Statement

We can use one **switch** inside another **switch**.

```
class Main {  
    public static void main(String[] args) {  
        int country = 1; // 1 - India, 2 - USA  
        int state = 2; // 1 - Maharashtra, 2 - Karnataka  
  
        switch (country) {  
            case 1:
```

```
System.out.println("Country: India");
switch (state) {
    case 1:
        System.out.println("State: Maharashtra");
        break;
    case 2:
        System.out.println("State: Karnataka");
        break;
    default:
        System.out.println("Unknown state");
}
break;
case 2:
    System.out.println("Country: USA");
    break;
default:
    System.out.println("Unknown country");
}
}
```

Expected Output

Country: India
State: Karnataka

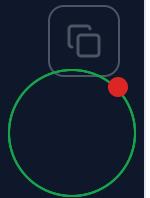


Enhanced switch in Java 12+

From Java 12 onwards, **switch** is enhanced with a new syntax.

Example 6: Enhanced switch with >

```
class Main {
    public static void main(String[] args) {
        int day = 4;
```



```
String result = switch (day) {  
    case 1, 2, 3 -> "Weekday";  
    case 4, 5 -> "Almost Weekend";  
    case 6, 7 -> "Weekend";  
    default -> "Invalid day";  
};  
  
System.out.println(result);  
}  
`
```

Expected Output

Almost Weekend



Explanation:

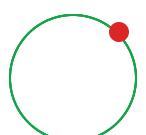
- The `switch` returns a value directly.
- We use `>` instead of `:` to make the code cleaner.

Rules and Limitations of switch

- Expression type: `switch` works with `byte`, `short`, `char`, `int`, `String`, and `enum`.
- No `long`, `float`, `double`, or `boolean` values allowed.
- Duplicate case values are not allowed.
- Use `break` to prevent fall-through.
- `default` is optional but recommended.

Conclusion

In this blog, we have learned:



Java Tutorial – A Comprehensive Guide for Beginners

Introduction

Java is one of the most widely used and versatile programming languages in the world. From desktop applications to web services and mobile apps to enterprise systems, Java powers countless technologies and platforms.

This Java tutorial is designed for beginners who want to build a strong foundation in Java programming. Whether you're completely new to programming or transitioning from another language, this guide covers everything from core concepts to essential object-oriented principles and beyond.

If you're aiming to become a proficient Java developer, follow this structured path to master Java, step by step.

Prerequisites

Before you dive into Java, it's helpful (but not mandatory) to be familiar with:

- **Basic Computer Knowledge:** Understanding how software runs, what an IDE is, etc.
- **Logic Building:** Basic problem-solving and logical thinking.



- **Understanding of Programming Concepts** (optional): Knowing variables, loops, or functions from another language is a bonus.

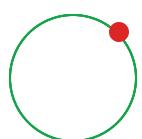
Master These Topics

Once you're ready, follow this structured list to learn Java from the ground up:

1. Introduction to Java
2. Java Fundamentals
3. Java Flow Control
4. Java Arrays
5. Java Methods
6. Java Object-Oriented Programming
7. Java Inner Classes
8. Static and Final Keywords
9. Java Exception Handling
10. Java Generics
11. Java Lambda Expressions
12. Java Collection Framework
13. Java Input/Output Streams
14. Java Reader/Writer

Introduction to Java

1. [Introduction to Java](#)
2. [Installing JDK on your computer](#)



3. Java Comments

Java Fundamentals

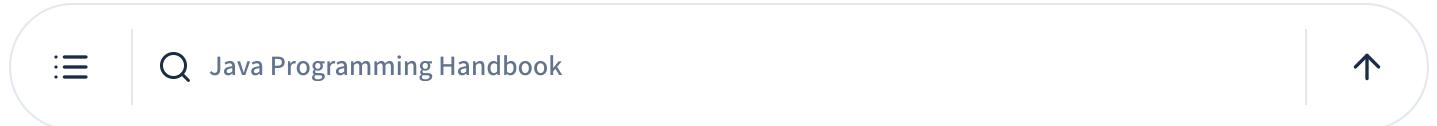
1. [Java Variables and Literals](#)
2. [Data Types in Java](#)
3. [Operators in Java](#)
4. [Java Basic Input and Output](#)
5. [Java Expressions, Statements and Blocks](#)

Java Flow Control

1. [Java if else statement](#)
2. [Java Ternary Operator](#)



Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#).

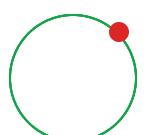


Java Arrays

1. [Java Arrays](#)
2. [Java Multidimensional Arrays](#)
3. [Java Copy Arrays](#)

Java Methods

1. [Methods in Java](#)



2. Parameter Passing in Java
3. Method Overloading
4. Variable Arguments
5. Java CommandLine Arguments
6. Java Recursive Method

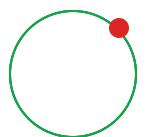
Java Object-Oriented Programming

1. Principles of Object-Oriented Programming
2. Java Class and Objects
3. Java Constructor
4. Encapsulation and Data Hiding
5. Inheritance in Java
6. Constructors in Inheritance
7. this Vs super Keyword
8. Method Overriding
9. Dynamic Method Dispatch
10. Polymorphism using Overloading and Overriding
11. Abstract Classes
12. Interfaces in Java

Java Inner Classes

1. Inner Classes in Java

Static and Final Keywords



1. Static Members and Static Blocks in Java
2. Final Members
3. Singleton Class

Java Exception Handling

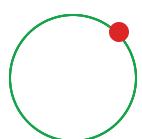
1. What are Exceptions
2. How to handle Exception
3. Try and Catch Block
4. Multiple and Nested Try Catch
5. Class Exception
6. Checked and Unchecked Exceptions
7. Throw Vs Throws
8. Finally Block
9. Try with Resources

Java Generics

1. Introduction to Generics
2. Defining Generic Class
3. Bounds on Generics
4. Java Generic Methods

Java Lambda Expressions

1. Introduction to Lambda Expressions
2. Parameters in Lambda Expressions



3. Capture in Lambda Expression

4. Method References

Java Collection Framework

1. Collections in Java

2. Understanding Collection Interface

3. Understanding List Interface in Java

4. Java ArrayList

5. Java LinkedList

6. Java Vector

7. Java Stack

8. Understanding Queue Interface in Java

9. Java PriorityQueue

10. Java ArrayDeque

11. Understanding Set Interface in Java

12. Java HashSet

13. Java LinkedHashSet

14. Java TreeSet

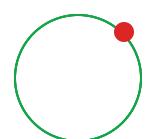
15. Understanding Map Interface in Java

16. Java HashMap

17. Java LinkedHashMap

18. Java TreeMap

19. Comparable Interface in Java



20. Comparator Interface in Java

Java Input/Output Streams

1. [What are Streams](#)
2. [InputStream and OutputStream Classes](#)
3. [FileInputStream and FileOutputStream](#)
4. [ByteArrayInputStream and ByteArrayOutputStream](#)
5. [ObjectInputStream and ObjectOutputStream](#)
6. [BufferedInputStream and BufferedOutputStream](#)
7. [Java PrintStream Class](#)

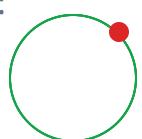
Java Reader/Writer

1. [Java File Class](#)
2. [Java Reader and Writer Class](#)
3. [InputStreamReader and OutputStreamWriter Class](#)
4. [FileReader and FileWriter Class](#)
5. [BufferedReader and BufferedWriter Class](#)
6. [StringReader and StringWriter Class](#)
7. [Java PrintWriter Class](#)

Key Features of Java

Java has remained popular for decades for good reason. Here's what makes it special:

- **Platform Independent:** Write once, run anywhere (thanks to the JVM).





 Pay Day Sale is Now Live!  Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



 Java Programming Handbook



Java Arrays

An array in Java is a data structure that allows us to store multiple values of the same type in a single variable. Instead of declaring separate variables for each value, we can use an array to efficiently manage data.

In this blog, we will cover:

- What is an Array?
- How to Declare an Array
- How to Initialize an Array
- Accessing and Modifying Array Elements
- Array Length
- Looping Through an Array
- Example Programs with Explanations
- Limitations of Arrays

What is an Array?

An array is a collection of elements stored **contiguously** in memory. All elements in an array must be of the **same data type**.



For example, if we want to store the marks of 5 students, instead of declaring multiple variables:

```
int mark1, mark2, mark3, mark4, mark5;
```



We can use an array:

```
int[] marks = new int[5];
```



This creates an array that can store 5 integer values.

How to Declare an Array

In Java, we can declare an array in two ways:

1. Using the new keyword (Preferred)

```
int[] numbers = new int[5]; // Declares an array of size 5
```

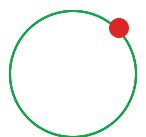


What happens when you declare an array with the `new` keyword:

- You are telling Java to **create a new array object** of type `int` and size `5`.
- Since `int` is a primitive type, all elements in the array are **automatically initialized to `0`** (the default value for `int`).

Internally:

- Memory is allocated for 5 `int`s.
- Values are: `[0, 0, 0, 0, 0]`.



2. Using Direct Initialization

```
int[] numbers = {10, 20, 30, 40, 50}; // Array with 5 elements
```



What's going on:

- This is a **shorthand syntax** for declaring and initializing the array in one step.
- Java **automatically infers** the size of the array based on the number of values provided (**5** here).

Internally:

- A new array of size 5 is created **and filled immediately** with the given values.

How to Initialize an Array

We can initialize an array in different ways:

1. Default Initialization

When an array is declared using **new**, its elements are automatically initialized to:

- **0** for numeric types (**int**, **double**, **float**, etc.).
- **false** for **boolean** type.
- **null** for object types (**String**, **Integer**, etc.).

```
class Main {
    public static void main(String[] args) {
        int[] numbers = new int[5]; // All elements initialized to 0
        // Print default values
        System.out.println(numbers[0]); // Output: 0
    }
}
```



```
}
```

Output:

```
0
```

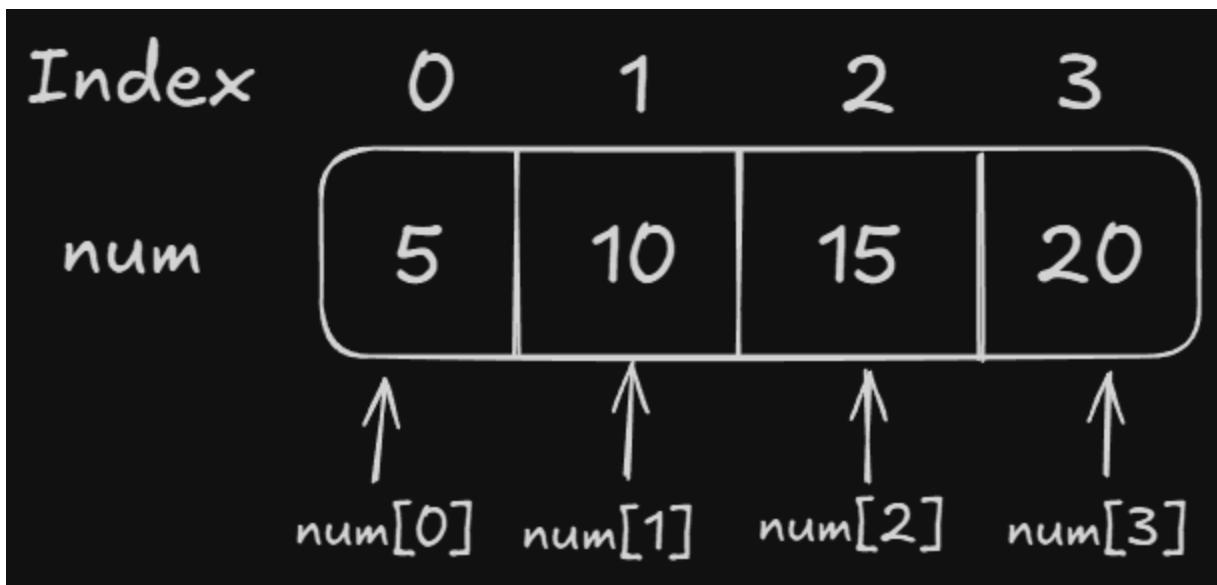


2. Explicit Initialization

```
int[] numbers = {10, 20, 30, 40, 50}; // Elements explicitly initialized
```



Accessing and Modifying Array Elements



Array elements are accessed using **index numbers**, starting from 0.

```
class Main {
    public static void main(String[] args) {
        int[] num = {5, 10, 15, 20};

        // Accessing elements
        System.out.println("First element: " + num[0]);
```



```
// Modifying elements  
num[1] = 50;  
System.out.println("Updated second element: " + num[1]);  
}  
}
```

Output:

```
First element: 5  
Updated second element: 50
```

Array Length

To find the number of elements in an array, we use `.length`.

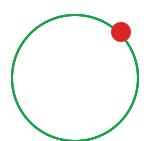
```
class Main {  
    public static void main(String[] args) {  
        int[] numbers = {1, 2, 3, 4, 5};  
  
        // Get array length  
        System.out.println("Array Length: " + numbers.length);  
    }  
}
```

Output:

```
Array Length: 5
```

Looping Through an Array

We can iterate over an array using:



1. for Loop

```
class Main {  
    public static void main(String[] args) {  
        int[] numbers = {10, 20, 30, 40, 50};  
  
        for (int i = 0; i < numbers.length; i++) {  
            System.out.println("Element at index " + i + ": " + numbers[i]);  
        }  
    }  
}
```

Output:

```
Element at index 0: 10  
Element at index 1: 20  
Element at index 2: 30  
Element at index 3: 40  
Element at index 4: 50
```

2. Enhanced for Loop (for-each)

A simpler way to iterate through an array is using the **enhanced for loop**.

```
class Main {  
    public static void main(String[] args) {  
        int[] numbers = {10, 20, 30, 40, 50};  
  
        for (int num : numbers) {  
            System.out.println("Number: " + num);  
        }  
    }  
}
```

Output:

```
Number: 10
Number: 20
Number: 30
Number: 40
Number: 50
```



Example Programs

Example 1: Find the Largest Element in an Array

```
class Main {
    public static void main(String[] args) {
        int[] numbers = {15, 42, 7, 98, 30};
        int max = numbers[0];

        for (int i = 1; i < numbers.length; i++) {
            if (numbers[i] > max) {
                max = numbers[i];
            }
        }

        System.out.println("Largest Number: " + max);
    }
}
```



Output:

```
Largest Number: 98
```



Example 2: Calculate the Sum of Array Elements

```
class Main {
    public static void main(String[] args) {
        int[] numbers = {5, 10, 15, 20, 25};
```



```
int sum = 0;

for (int num : numbers) {
    sum += num;
}

System.out.println("Sum of Elements: " + sum);
}
}
```

Output:

```
Sum of Elements: 75
```



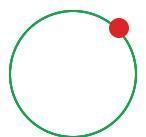
Limitations of Arrays

1. **Fixed Size:** Once declared, an array's size **cannot be changed**.
2. **Same Data Type:** Arrays can only store elements of **one data type**.
3. **Inefficient Insertion/Deletion:** Adding/removing elements in the middle of an array is **slow**.

Conclusion

In this blog, we have learned:

- **What arrays are** and why they are useful.
- **How to declare and initialize arrays** in Java.
- **How to access and modify array elements**.
- **Finding the length of an array** using `.length`.





 Pay Day Sale is Now Live!  Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



 Java Programming Handbook



Java Multidimensional Arrays

In Java, arrays can have **multiple dimensions**. A **multidimensional array** is an array of arrays, meaning each element of the main array can be another array. This is useful when dealing with **tabular data, matrices, or grids**.

In this blog, we will cover:

- What is a Multidimensional Array?
- How to Declare a Multidimensional Array
- How to Initialize a Multidimensional Array
- Accessing and Modifying Elements
- Looping Through a Multidimensional Array
- Example Programs with Expected Outputs
- Limitations of Multidimensional Arrays

What is a Multidimensional Array?

A **multidimensional array** is an array where each element is itself an array. The most commonly used multidimensional array is a **two-dimensional array (2D array)**.

Example of a 2D array:



```
1 2 3  
4 5 6  
7 8 9
```



Here, each **row** is an array containing multiple elements.

How to Declare a Multidimensional Array

In Java, we declare a **2D array** as follows:

```
dataType[][] arrayName;
```



For example:

```
int[][] numbers;
```



We can also declare **higher-dimensional arrays** (3D, 4D, etc.), but 2D is the most commonly used.

How to Initialize a Multidimensional Array

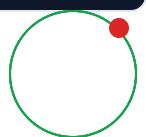
We can initialize a **2D array** in multiple ways.

1. Using new Keyword

```
// Declaring and initializing a 3x3 matrix  
int[][] numbers = new int[3][3]; // Creates a 3x3 array
```



This creates a **3x3** array with default values (**0** for integers).



2. Direct Initialization

```
// Initializing a 2x3 array
int[][] numbers = {
    {1, 2, 3},
    {4, 5, 6}
};
```

Here:

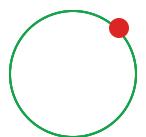
- `numbers[0] = {1, 2, 3}`
- `numbers[1] = {4, 5, 6}`

Accessing and Modifying Elements

	Column-1	Column-2	Column-3
Row-1	num[0][0]	num[0][1]	num[0][2]
Row-2	num[1][0]	num[1][1]	num[1][2]
Row-3	num[2][0]	num[2][1]	num[2][2]

Java Multidimensional Array

To access elements, we use **row index** and **column index**.



```
class Main {  
    public static void main(String[] args) {  
        int[][] num = {  
            {10, 20, 30},  
            {40, 50, 60},  
            {70, 80, 90}  
        };  
  
        // Accessing elements  
        System.out.println("Element at row 1, column 2: " + num[1][2]); // Output  
  
        // Modifying elements  
        num[0][1] = 25;  
        System.out.println("Updated element at row 0, column 1: " + num[0][1]);  
    }  
}
```

Output: 25

Looping Through a Multidimensional Array

We can iterate through a 2D array using **nested loops**.

1. Using for Loop

```
class Main {  
    public static void main(String[] args) {  
        int[][] numbers = {  
            {1, 2, 3},  
            {4, 5, 6}  
        };  
  
        // Loop through rows  
        for (int i = 0; i < numbers.length; i++) {  
            // Loop through columns  
            for (int j = 0; j < numbers[i].length; j++) {  
                System.out.println(numbers[i][j]);  
            }  
        }  
    }  
}
```



```
        System.out.print(numbers[i][j] + " ");
    }
    System.out.println(); // Move to the next row
}
}
```

Output

```
1 2 3
4 5 6
```



2. Using Enhanced for Loop

```
class Main {
    public static void main(String[] args) {
        int[][] numbers = {
            {10, 20, 30},
            {40, 50, 60}
        };

        for (int[] row : numbers) {
            for (int num : row) {
                System.out.print(num + " ");
            }
            System.out.println();
        }
    }
}
```



Output

```
10 20 30
40 50 60
```



Example Programs

Example 1: Sum of All Elements in a 2D Array

```
class Main {  
    public static void main(String[] args) {  
        int[][] numbers = {  
            {1, 2, 3},  
            {4, 5, 6}  
        };  
  
        int sum = 0;  
  
        for (int i = 0; i < numbers.length; i++) {  
            for (int j = 0; j < numbers[i].length; j++) {  
                sum += numbers[i][j]; // Add each element to sum  
            }  
        }  
  
        System.out.println("Sum of all elements: " + sum);  
    }  
}
```

Output

```
Sum of all elements: 21
```

Example 2: Find Maximum Element in a 2D Array

```
class Main {  
    public static void main(String[] args) {  
        int[][] numbers = {  
            {3, 5, 7},  
            {9, 2, 8}  
        };  
    }  
}
```

```
int max = numbers[0][0];

for (int i = 0; i < numbers.length; i++) {
    for (int j = 0; j < numbers[i].length; j++) {
        if (numbers[i][j] > max) {
            max = numbers[i][j]; // Update max if a larger element is found
        }
    }
}

System.out.println("Maximum element: " + max);
}
```

Output

```
Maximum element: 9
```



Jagged Arrays in Java

A **jagged array** is a multidimensional array where **each row can have a different number of columns**.

Example:

```
class Main {
    public static void main(String[] args) {
        // Declaring a jagged array
        int[][] jaggedArray = {
            {1, 2, 3},
            {4, 5},
            {6, 7, 8, 9}
        };

        for (int i = 0; i < jaggedArray.length; i++) {
            for (int j = 0; j < jaggedArray[i].length; j++) {
```



```
        System.out.print(jaggedArray[i][j] + " ");
    }
    System.out.println();
}
}
```

Output

```
1 2 3
4 5
6 7 8 9
```



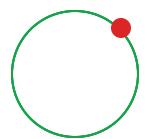
Limitations of Multidimensional Arrays

1. **Increased Memory Usage:** Multidimensional arrays require more memory.
2. **Difficult to Manage:** Nested loops make it more complex to work with.
3. **Fixed Size:** Once declared, the size **cannot** be changed.

Conclusion

In this blog, we learned:

- What **multidimensional arrays** are and why they are useful.
- How to **declare, initialize, and access** elements in a **2D array**.
- How to **loop through** a multidimensional array using **for** and **for-each** loops.
- **Example programs:** Finding sum and maximum elements in a 2D array.
- Introduction to **jagged arrays**.
- Limitations of using multidimensional arrays.





 Pay Day Sale is Now Live!  Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



 Java Programming Handbook



Java Copy Arrays

In Java, sometimes we need to **copy an array** into another array. This is useful when we want to create a duplicate of an existing array without modifying the original one.

In this blog, we will cover:

Different ways to copy an array

- Using **Loop** for copying
- Using **arraycopy()** method
- Using **clone()** method
- Using **Arrays.copyOf()** method
- Using **Arrays.copyOfRange()** method

Example programs with expected outputs

1. Copying an Array Using a Loop

The most basic way to copy an array is to use a **loop**.

Example: Copying an array using a loop





```
class Main {  
    public static void main(String[] args) {  
        // Original array  
        int[] original = {10, 20, 30, 40, 50};  
  
        // New array of the same size  
        int[] copied = new int[original.length];  
  
        // Copy elements using a loop  
        for (int i = 0; i < original.length; i++) {  
            copied[i] = original[i];  
        }  
  
        // Printing copied array  
        System.out.print("Copied Array: ");  
        for (int num : copied) {  
            System.out.print(num + " ");  
        }  
    }  
}
```

Output:



```
Copied Array: 10 20 30 40 50
```

Best For: Small-sized arrays when performance is not a major concern.

2. Copying an Array Using `System.arraycopy()`

The `System.arraycopy()` method is an efficient way to copy an array in Java.

Syntax:



```
System.arraycopy(sourceArray, sourcePosition, destinationArray, destination
```

- **sourceArray** → The original array
- **sourcePosition** → The starting index to copy from
- **destinationArray** → The new array where values will be copied
- **destinationPosition** → The index where copying starts in the destination array
- **length** → Number of elements to copy

Example: Using System.arraycopy()

```
class Main {
    public static void main(String[] args) {
        int[] original = {1, 2, 3, 4, 5};

        int[] copied = new int[original.length];

        // Copying using System.arraycopy
        System.arraycopy(original, 0, copied, 0, original.length);

        // Printing copied array
        System.out.print("Copied Array: ");
        for (int num : copied) {
            System.out.print(num + " ");
        }
    }
}
```

Output:

Copied Array: 1 2 3 4 5

Best For: Fast copying when working with large arrays.

3. Copying an Array Using clone()

The `clone()` method is used to create a **shallow copy** of an array. It works only for **one-dimensional arrays**.

Example: Using `clone()`

```
class Main {  
    public static void main(String[] args) {  
        int[] original = {7, 14, 21, 28, 35};  
  
        // Using clone() to copy array  
        int[] copied = original.clone();  
  
        // Printing copied array  
        System.out.print("Copied Array: ");  
        for (int num : copied) {  
            System.out.print(num + " ");  
        }  
    }  
}
```

Output:

```
Copied Array: 7 14 21 28 35
```

Best For: Quick duplication of arrays when dealing with **primitive types**.

Limitation: If the array contains objects, `clone()` performs **shallow copying**, meaning it only copies object references, not the actual objects.

4. Copying an Array Using `Arrays.copyOf()`

The `Arrays.copyOf()` method is a simple way to copy an array while also allowing us to change its size.

Example: Using Arrays.copyOf()

```
import java.util.Arrays;  
  
class Main {  
    public static void main(String[] args) {  
        int[] original = {5, 10, 15, 20, 25};  
  
        // Copying entire array  
        int[] copied = Arrays.copyOf(original, original.length);  
  
        // Printing copied array  
        System.out.print("Copied Array: ");  
        for (int num : copied) {  
            System.out.print(num + " ");  
        }  
    }  
}
```

Expected Output:

```
Copied Array: 5 10 15 20 25
```

Best For: Copying an entire array or resizing an array during copying.

5. Copying a Portion of an Array Using Arrays.copyOfRange()

The `Arrays.copyOfRange()` method is useful when we need to copy only a specific range of elements.

Syntax:

```
Arrays.copyOfRange(sourceArray, fromIndex, toIndex);
```

- `fromIndex` → Start index (inclusive)
- `toIndex` → End index (exclusive)

Example: Copying a Range of an Array

```
import java.util.Arrays;  
  
class Main {  
    public static void main(String[] args) {  
        int[] original = {2, 4, 6, 8, 10, 12};  
  
        // Copying elements from index 2 to 5 (excluding 5)  
        int[] copied = Arrays.copyOfRange(original, 2, 5);  
  
        // Printing copied array  
        System.out.print("Copied Array: ");  
        for (int num : copied) {  
            System.out.print(num + " ");  
        }  
    }  
}
```

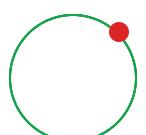
Expected Output:

```
Copied Array: 6 8 10
```

Best For: Extracting a sub-array from a larger array.

Choosing the Right Method to Copy an Array

Method	When to Use?
Loop Method	For small arrays, simple approach



Java Tutorial – A Comprehensive Guide for Beginners

Introduction

Java is one of the most widely used and versatile programming languages in the world. From desktop applications to web services and mobile apps to enterprise systems, Java powers countless technologies and platforms.

This Java tutorial is designed for beginners who want to build a strong foundation in Java programming. Whether you're completely new to programming or transitioning from another language, this guide covers everything from core concepts to essential object-oriented principles and beyond.

If you're aiming to become a proficient Java developer, follow this structured path to master Java, step by step.

Prerequisites

Before you dive into Java, it's helpful (but not mandatory) to be familiar with:

- **Basic Computer Knowledge:** Understanding how software runs, what an IDE is, etc.
- **Logic Building:** Basic problem-solving and logical thinking.



- **Understanding of Programming Concepts** (optional): Knowing variables, loops, or functions from another language is a bonus.

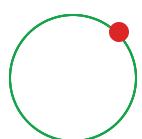
Master These Topics

Once you're ready, follow this structured list to learn Java from the ground up:

1. Introduction to Java
2. Java Fundamentals
3. Java Flow Control
4. Java Arrays
5. Java Methods
6. Java Object-Oriented Programming
7. Java Inner Classes
8. Static and Final Keywords
9. Java Exception Handling
10. Java Generics
11. Java Lambda Expressions
12. Java Collection Framework
13. Java Input/Output Streams
14. Java Reader/Writer

Introduction to Java

1. [Introduction to Java](#)
2. [Installing JDK on your computer](#)



3. Java Comments

Java Fundamentals

1. [Java Variables and Literals](#)
2. [Data Types in Java](#)
3. [Operators in Java](#)
4. [Java Basic Input and Output](#)
5. [Java Expressions, Statements and Blocks](#)

Java Flow Control

1. [Java if else statement](#)
2. [Java Ternary Operator](#)
3. [Java For Loop](#)
4. [Java while and do while loop](#)
5. [Java continue and break statement](#)
6. [Java Switch statement](#)



Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)

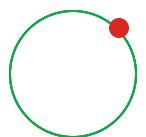


Java Programming Handbook



Java Methods

1. [Methods in Java](#)



2. Parameter Passing in Java
3. Method Overloading
4. Variable Arguments
5. Java CommandLine Arguments
6. Java Recursive Method

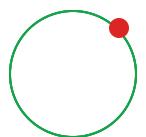
Java Object-Oriented Programming

1. Principles of Object-Oriented Programming
2. Java Class and Objects
3. Java Constructor
4. Encapsulation and Data Hiding
5. Inheritance in Java
6. Constructors in Inheritance
7. this Vs super Keyword
8. Method Overriding
9. Dynamic Method Dispatch
10. Polymorphism using Overloading and Overriding
11. Abstract Classes
12. Interfaces in Java

Java Inner Classes

1. Inner Classes in Java

Static and Final Keywords



1. Static Members and Static Blocks in Java
2. Final Members
3. Singleton Class

Java Exception Handling

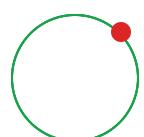
1. What are Exceptions
2. How to handle Exception
3. Try and Catch Block
4. Multiple and Nested Try Catch
5. Class Exception
6. Checked and Unchecked Exceptions
7. Throw Vs Throws
8. Finally Block
9. Try with Resources

Java Generics

1. Introduction to Generics
2. Defining Generic Class
3. Bounds on Generics
4. Java Generic Methods

Java Lambda Expressions

1. Introduction to Lambda Expressions
2. Parameters in Lambda Expressions



3. Capture in Lambda Expression

4. Method References

Java Collection Framework

1. Collections in Java

2. Understanding Collection Interface

3. Understanding List Interface in Java

4. Java ArrayList

5. Java LinkedList

6. Java Vector

7. Java Stack

8. Understanding Queue Interface in Java

9. Java PriorityQueue

10. Java ArrayDeque

11. Understanding Set Interface in Java

12. Java HashSet

13. Java LinkedHashSet

14. Java TreeSet

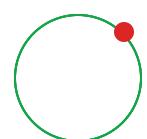
15. Understanding Map Interface in Java

16. Java HashMap

17. Java LinkedHashMap

18. Java TreeMap

19. Comparable Interface in Java



20. Comparator Interface in Java

Java Input/Output Streams

1. [What are Streams](#)
2. [InputStream and OutputStream Classes](#)
3. [FileInputStream and FileOutputStream](#)
4. [ByteArrayInputStream and ByteArrayOutputStream](#)
5. [ObjectInputStream and ObjectOutputStream](#)
6. [BufferedInputStream and BufferedOutputStream](#)
7. [Java PrintStream Class](#)

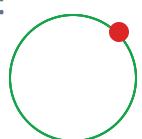
Java Reader/Writer

1. [Java File Class](#)
2. [Java Reader and Writer Class](#)
3. [InputStreamReader and OutputStreamWriter Class](#)
4. [FileReader and FileWriter Class](#)
5. [BufferedReader and BufferedWriter Class](#)
6. [StringReader and StringWriter Class](#)
7. [Java PrintWriter Class](#)

Key Features of Java

Java has remained popular for decades for good reason. Here's what makes it special:

- **Platform Independent:** Write once, run anywhere (thanks to the JVM).





 Pay Day Sale is Now Live!  Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



 Java Programming Handbook



Methods in Java

In Java, **methods** are blocks of code that perform a specific task. Instead of writing the same code multiple times, we can define a method once and call it whenever needed.

Using methods helps in:

- Making the code more **organized** and **readable**
- Avoiding **code repetition**
- Improving **code reusability**

In this blog, we will cover:

- What is a Method in Java?
- How to Declare a Method
- Calling a Method in Java
- Example Programs

1. What is a Method in Java?

A **method** is a group of statements that perform a specific operation. A method is defined once but can be called multiple times.



Example of a Simple Method:

```
class Main {  
    // Method to display a message  
    static void greet() {  
        System.out.println("Hello! Welcome to Java.");  
    }  
  
    public static void main(String[] args) {  
        // Calling the method  
        greet();  
    }  
}
```

Expected Output:

```
Hello! Welcome to Java.
```

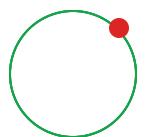
2. How to Declare a Method in Java?

A method in Java is declared using the following syntax:

```
returnType methodName(parameters) {  
    // method body  
    return value; // (only if returnType is not void)  
}
```

Explanation of Method Components:

- **returnType** → The type of value the method returns. If it does not return anything, use **void**.
- **methodName** → The name of the method (should be meaningful).



- **parameters** → Input values passed to the method (optional).
- **method body** → The set of statements that define what the method does.

3. Calling a Method in Java

A method must be **called** to execute its code.

Example: Calling a Method

```
class Main {  
    // Method that prints a message  
    static void displayMessage() {  
        System.out.println("Java is fun!");  
    }  
  
    public static void main(String[] args) {  
        // Calling the method  
        displayMessage();  
    }  
}
```

Expected Output:

Java is fun!

4. Example: Method with Parameters and Return Value

A method can take **parameters** (input values) and return a value using the **return** statement.

Example: Adding Two Numbers Using a Method

```
class Main {  
    // Method to add two numbers and return the sum  
    static int addNumbers(int a, int b) {  
        return a + b;  
    }  
  
    public static void main(String[] args) {  
        // Calling the method and storing the result  
        int sum = addNumbers(10, 20);  
  
        // Printing the result  
        System.out.println("Sum: " + sum);  
    }  
}
```

Expected Output:

Sum: 30



5. Method Returning No Value (void Method)

If a method does not return any value, we use **void**.

Example: Printing a Message Using a Void Method

```
class Main {  
    // Method with no return value  
    static void printWelcome() {  
        System.out.println("Welcome to Java Programming!");  
    }  
  
    public static void main(String[] args) {  
        // Calling the method  
        printWelcome();  
    }  
}
```



Expected Output:

Welcome to Java Programming!



6. Using Methods with Different Return Types

A method can return **different data types**, such as `int`, `double`, `String`, etc.

Example: Returning a String from a Method

```
class Main {  
    // Method returning a String  
    static String getMessage() {  
        return "Hello from the getMessage method!";  
    }  
  
    public static void main(String[] args) {  
        // Calling the method and printing the result  
        System.out.println(getMessage());  
    }  
}
```



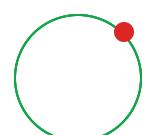
Expected Output:

Hello from the getMessage method!



7. Method with Multiple Parameters

A method can take **multiple parameters** separated by commas.



Example: Multiplication of Two Numbers Using a Method

```
class Main {  
    // Method to multiply two numbers  
    static int multiply(int x, int y) {  
        return x * y;  
    }  
  
    public static void main(String[] args) {  
        // Calling the method with arguments  
        int result = multiply(5, 4);  
  
        // Printing the result  
        System.out.println("Multiplication Result: " + result);  
    }  
}
```

Expected Output:

```
Multiplication Result: 20
```

8. Calling a Method Multiple Times

A method can be called multiple times to perform the same task.

Example: Reusing a Method

```
class Main {  
    // Method to display a message  
    static void sayHello() {  
        System.out.println("Hello!");  
    }  
  
    public static void main(String[] args) {  
        // Calling the method multiple times  
    }  
}
```



 Pay Day Sale is Now Live!  Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



 Java Programming Handbook



Parameter Passing in Java

In Java, we often pass values to methods so they can perform operations using those values.

This is called **parameter passing**. Java supports two types of parameter passing:

- **Pass-by-Value (Primitive Data Types)**
- **Pass-by-Reference (Objects & Arrays)**

In this blog, we will cover:

- How parameters work in Java
- Passing **primitive data types** to methods
- Passing **objects** to methods
- Passing **arrays** to methods
- Key differences between **pass-by-value** and **pass-by-reference**

1. How Parameters Work in Java?

In Java, **all method arguments are passed by value**, meaning a **copy** of the value is passed to the method. However, how this affects the original variable depends on whether we pass:

- Primitive data types (like `int`, `double`) → Original value does not change



- Objects & arrays → Original object can be modified

2. Passing Primitive Data Types (Pass-by-Value)

When we pass a primitive data type (like `int`, `double`, `char`) to a method, only a **copy** of the value is passed. Changes inside the method do **not** affect the original value.

Example: Passing Primitive Data Types

```
class Example {
    // Method that tries to modify the value
    static void modifyValue(int num) {
        num = num + 10; // Change the value
        System.out.println("Inside Method: " + num);
    }

    public static void main(String[] args) {
        int number = 50;

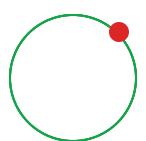
        // Passing primitive type to method
        modifyValue(number);

        // Original value remains unchanged
        System.out.println("Outside Method: " + number);
    }
}
```

Output:

```
Inside Method: 60
Outside Method: 50
```

Explanation:

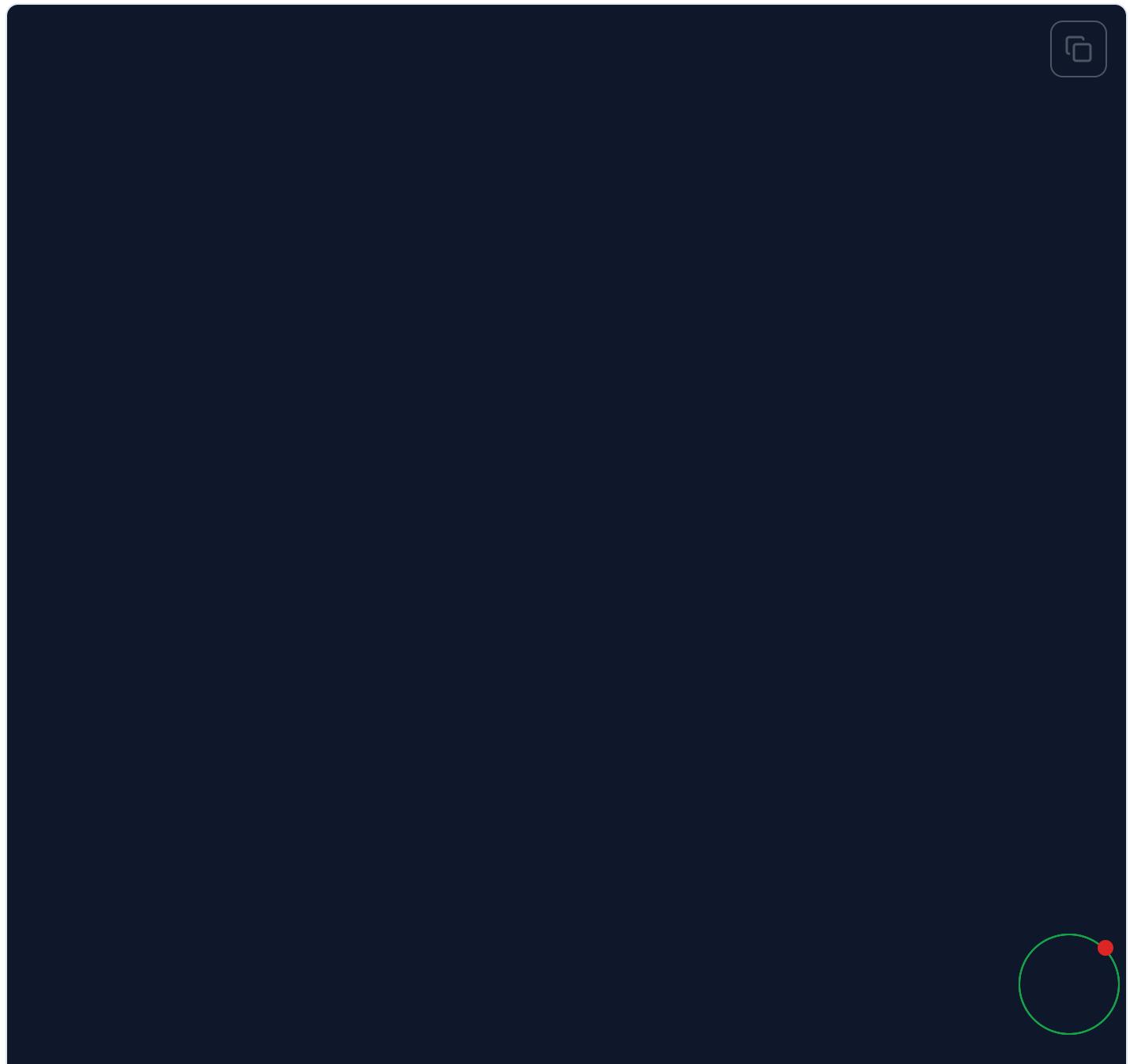


- Inside the method, `num` is changed, but the original `number` remains **unchanged** because only a **copy** was passed.

3. Passing Objects (Pass-by-Reference)

When we pass an **object** to a method, a **copy of the reference** (memory address) is passed. This means **changes made inside the method will affect the original object**.

Example: Passing Objects to Methods



```

class Person {
    String name;

    // Constructor
    Person(String name) {
        this.name = name;
    }

    // Method that modifies the object
    static void changeName(Person p) {
        p.name = "Updated Name"; // Modifying object
    }

    public static void main(String[] args) {
        // Creating an object
        Person person1 = new Person("John");

        // Printing before modification
        System.out.println("Before: " + person1.name);

        // Passing object to method
        changeName(person1);

        // Printing after modification
        System.out.println("After: " + person1.name);
    }
}

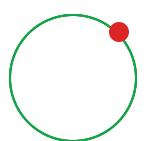
```

Output:

Before: John
After: Updated Name

Explanation:

- The **Person** object is passed to the method, and its **name** property is changed.



- Since objects are passed by reference, the original object is modified.

4. Passing Arrays to Methods

Like objects, arrays are also passed by reference, meaning changes inside the method will affect the original array.

Example: Passing an Array to a Method

```
class Example {  
    // Method that modifies an array  
    static void modifyArray(int[] arr) {  
        arr[0] = 100; // Changing the first element  
    }  
  
    public static void main(String[] args) {  
        int[] numbers = {1, 2, 3, 4};  
  
        // Printing before modification  
        System.out.println("Before: " + numbers[0]);  
  
        // Passing array to method  
        modifyArray(numbers);  
  
        // Printing after modification  
        System.out.println("After: " + numbers[0]);  
    }  
}
```

Output:

```
Before: 1  
After: 100
```

Explanation:

- The `modifyArray()` method changes the first element of the array.
- Since arrays are **passed by reference**, the original array is modified.

5. Key Differences: Pass-by-Value vs Pass-by-Reference

Feature	Primitive Data Types (int, double, char)	Objects & Arrays
Pass Type	Pass-by-Value (Copy is Passed)	Pass-by-Reference (Reference is Passed)
Modification inside Method?	No (Original value remains same)	Yes (Original object is modified)
Effect on Original Data	No effect	Changes are reflected

6. Returning Values from a Method

A method can also **return** modified data.

Example: Returning a Modified Value

```
class Example {
    // Method that returns a new value
    static int addTen(int num) {
        return num + 10;
    }

    public static void main(String[] args) {
        int number = 50;

        // Storing the returned value
        number = addTen(number);

        System.out.println("Updated Value: " + number);
    }
}
```





 Pay Day Sale is Now Live!  Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



 Java Programming Handbook



Method Overloading in Java

In Java, **Method Overloading** allows us to define multiple methods with the **same name** but different parameters. It is an example of **compile-time polymorphism**, where the correct method is selected at **compile time** based on the **method signature**.

Why Use Method Overloading?

Method overloading helps in:

- Improving **code readability** and **reusability**
- Making the **code cleaner** and more **maintainable**
- Reducing redundancy by allowing multiple methods with the same name but different parameters

Rules for Method Overloading

To overload a method in Java, you must follow these rules:

- **Same method name but different parameters** (different number, type, or order of parameters)
- **Return type does NOT matter** for method overloading (only parameters matter)



- Methods can have different access modifiers

1. Method Overloading with Different Number of Parameters

We can overload a method by changing the **number of parameters**.

Example: Different Number of Parameters

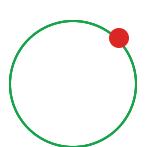
```
class Calculator {  
    // Method with two parameters  
    static int add(int a, int b) {  
        return a + b;  
    }  
  
    // Overloaded method with three parameters  
    static int add(int a, int b, int c) {  
        return a + b + c;  
    }  
  
    public static void main(String[] args) {  
        // Calling methods with different arguments  
        System.out.println("Sum of 2 numbers: " + add(10, 20));  
        System.out.println("Sum of 3 numbers: " + add(10, 20, 30));  
    }  
}
```

Output:

```
Sum of 2 numbers: 30  
Sum of 3 numbers: 60
```

Explanation:

- `add(int, int)` is called when **two** arguments are passed.



- `add(int, int, int)` is called when **three** arguments are passed.
- The correct method is selected based on the **number of parameters**.

2. Method Overloading with Different Data Types

We can also overload methods by changing the **data type** of parameters.

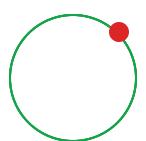
Example: Different Data Types

```
class Display {  
    // Method with an int parameter  
    static void show(int num) {  
        System.out.println("Integer: " + num);  
    }  
  
    // Overloaded method with a double parameter  
    static void show(double num) {  
        System.out.println("Double: " + num);  
    }  
  
    public static void main(String[] args) {  
        // Calling methods with different types of arguments  
        show(10);          // Calls the int version  
        show(10.5);        // Calls the double version  
    }  
}
```

Output:

```
Integer: 10  
Double: 10.5
```

Explanation:



- `show(int)` is called when an **integer** is passed.
- `show(double)` is called when a **double** is passed.
- The correct method is selected based on the **type of parameter**.

3. Method Overloading with Different Order of Parameters

We can overload methods by **changing the order of parameters**.

Example: Different Order of Parameters

```
class Printer {
    // Method with (String, int)
    static void printDetails(String name, int age) {
        System.out.println("Name: " + name + ", Age: " + age);
    }

    // Overloaded method with (int, String)
    static void printDetails(int age, String name) {
        System.out.println("Age: " + age + ", Name: " + name);
    }

    public static void main(String[] args) {
        // Calling methods with different order of parameters
        printDetails("Alice", 25);
        printDetails(30, "Bob");
    }
}
```

Output:

```
Name: Alice, Age: 25
Age: 30, Name: Bob
```

Explanation:

- Java considers `(String, int)` and `(int, String)` as different method signatures.
- The correct method is called based on the **order of parameters**.

4. Method Overloading with Different Return Types (Not Allowed Alone)

In Java, we **cannot overload methods by changing only the return type**. The compiler does **not differentiate** methods based on return type alone.

Example: This Will NOT Work

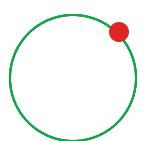
```
class Example {  
    // Method returning int  
    int add(int a, int b) {  
        return a + b;  
    }  
  
    // Overloaded method with same parameters but different return type (INVALID)  
    double add(int a, int b) {    // Compilation Error  
        return a + b;  
    }  
}
```

Error Message:

```
Error: method add(int,int) is already defined in class Example
```

Solution:

To overload methods, **parameters must be different**, not just the return type.

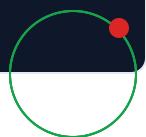


5. Constructor Overloading

Just like methods, **constructors** can also be overloaded. This allows us to create objects with different initial values.

Example: Constructor Overloading

```
class Student {  
    String name;  
    int age;  
  
    // Constructor with one parameter  
    Student(String name) {  
        this.name = name;  
        this.age = 18; // Default age  
    }  
  
    // Overloaded constructor with two parameters  
    Student(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    void display() {  
        System.out.println("Name: " + name + ", Age: " + age);  
    }  
  
    public static void main(String[] args) {  
        // Creating objects using different constructors  
        Student s1 = new Student("Alice");  
        Student s2 = new Student("Bob", 22);  
  
        s1.display();  
        s2.display();  
    }  
}
```



Output:

```
Name: Alice, Age: 18  
Name: Bob, Age: 22
```



Explanation:

- We created two constructors (`Student(String)` and `Student(String, int)`).
- Depending on the arguments, the correct constructor is selected.

Key Takeaways

Method Overloading allows multiple methods with the same name but different parameters.

Methods can be overloaded by changing:

1. Number of parameters
2. Data types of parameters
3. Order of parameters

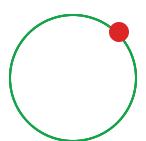
Return type alone does NOT differentiate overloaded methods.

Constructors can also be overloaded to initialize objects differently.

Conclusion

In this blog, we learned:

- What Method Overloading is and why it is useful





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



Variable Arguments (varargs) in Java

In Java, **variable arguments** (also called **varargs**) allow us to pass a **variable number of arguments** to a method. This feature simplifies method definitions by eliminating the need to define multiple overloaded methods for different argument counts.

Why Use Variable Arguments?

- Allows a method to accept any number of arguments
- Eliminates the need for method **overloading** in some cases
- Increases code **reusability** and reduces redundancy

Syntax of varargs

We use an **ellipsis** (`...`) after the data type to define a varargs method:

```
returnType methodName(dataType... variableName) { }
```



Example:

```
void displayNumbers(int... numbers) {  
    // Method body
```



Example: Using Variable Arguments

```
class VarargsExample {  
    // Method with variable arguments  
    static void printNumbers(int... numbers) {  
        System.out.println("Number of arguments: " + numbers.length);  
  
        // Loop through numbers and print each  
        for (int num : numbers) {  
            System.out.print(num + " ");  
        }  
        System.out.println();  
    }  
  
    public static void main(String[] args) {  
        // Calling method with different number of arguments  
        printNumbers(10, 20, 30);  
        printNumbers(5, 15);  
        printNumbers(100);  
    }  
}
```

Output:

```
Number of arguments: 3  
10 20 30  
Number of arguments: 2  
5 15  
Number of arguments: 1  
100
```

Explanation:

- The method `printNumbers(int... numbers)` can accept any number of integers.

- The `numbers.length` property helps in knowing the number of arguments passed.

Varargs with Other Parameters

A varargs parameter **must always be the last parameter** in the method signature.

Example: Valid Varargs Usage

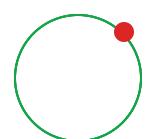
```
class Example {  
    static void showDetails(String name, int... marks) {  
        System.out.println("Student: " + name);  
        System.out.print("Marks: ");  
        for (int mark : marks) {  
            System.out.print(mark + " ");  
        }  
        System.out.println();  
    }  
  
    public static void main(String[] args) {  
        showDetails("Alice", 85, 90, 95);  
        showDetails("Bob", 80, 88);  
    }  
}
```

Output:

```
Student: Alice  
Marks: 85 90 95  
Student: Bob  
Marks: 80 88
```

Explanation:

- The `name` parameter is required, and varargs `marks` can take multiple values.





Pay Day Sale is Now Live! Use the Coupon PAYDAY and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Java Command-Line Arguments

Java allows us to pass **arguments** to the `main()` method when running a program from the command line. These arguments are received as a **String array** (`String[] args`).

Why Use Command-Line Arguments?

- Dynamic input without modifying the code
- Used in real-world applications to provide configuration settings
- Helpful in automation and scripting

How Command-Line Arguments Work?

When we run a Java program from the terminal, we can pass arguments after the class name:

```
java MyClass arg1 arg2 arg3
```



- These arguments are stored in the `String[] args` array.
- `args[0]` holds the first argument, `args[1]` holds the second, and so on.

Example: Printing Command-Line Arguments



```
class CommandLineExample {  
    public static void main(String[] args) {  
        System.out.println("Number of arguments: " + args.length);  
  
        // Print all arguments  
        for (int i = 0; i < args.length; i++) {  
            System.out.println("Argument " + (i + 1) + ": " + args[i]);  
        }  
    }  
}
```

How to Run in Terminal:

```
java CommandLineExample Hello World 123
```

Expected Output:

```
Number of arguments: 3  
Argument 1: Hello  
Argument 2: World  
Argument 3: 123
```

Explanation:

- The `args` array stores the command-line arguments.
- `args.length` gives the number of arguments passed.

Converting Command-Line Arguments to Other Data Types

Since command-line arguments are always strings, we may need to convert them.

Example: Adding Two Numbers from Command-Line Arguments

```
class SumCalculator {  
    public static void main(String[] args) {  
        if (args.length < 2) {  
            System.out.println("Please provide two numbers.");  
            return;  
        }  
  
        // Convert string arguments to integers  
        int num1 = Integer.parseInt(args[0]);  
        int num2 = Integer.parseInt(args[1]);  
  
        // Calculate and print the sum  
        System.out.println("Sum: " + (num1 + num2));  
    }  
}
```

How to Run in Terminal:

```
java SumCalculator 10 20
```

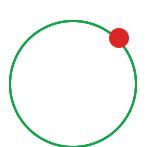
Expected Output:

```
Sum: 30
```

Explanation:

- `Integer.parseInt(args[0])` converts string arguments to integers.
- The program calculates and prints their sum.
- If less than two arguments are provided, it prints an error message.

Handling Errors in Command-Line Arguments



To avoid errors, always **validate** input before converting it.

Example: Handling Incorrect Input

```
class SafeConverter {  
    public static void main(String[] args) {  
        try {  
            if (args.length < 2) {  
                System.out.println("Please provide two numbers.");  
                return;  
            }  
  
            int num1 = Integer.parseInt(args[0]);  
            int num2 = Integer.parseInt(args[1]);  
  
            System.out.println("Sum: " + (num1 + num2));  
        } catch (NumberFormatException e) {  
            System.out.println("Invalid number format. Please enter valid integers.");  
        }  
    }  
}
```

How to Run in Terminal:

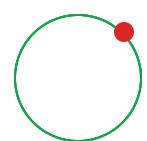
```
java SafeConverter 10 abc
```

Expected Output:

```
Invalid number format. Please enter valid integers.
```

Key Takeaways

- Command-line arguments are passed to `main(String[] args)`.





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Java Recursive Method

Introduction

A **recursive method** is a method that calls **itself** to solve a problem. Recursion is useful for problems that can be **broken down into smaller subproblems** of the same type. It is commonly used for tasks such as **factorial calculation**, **Fibonacci series**, **tree traversal**, and **searching algorithms**.

How Recursion Works in Java?

A recursive method must have:

1. **Base Case** – A condition that stops the recursion.
2. **Recursive Case** – A condition where the method **calls itself** with a modified argument to approach the base case.

Example: Simple Recursion

Let's start with a simple example of printing numbers from **n** to 1 using recursion.

```
class RecursionExample {  
    // Recursive method to print numbers
```



```

static void printNumbers(int n) {
    if (n == 0) { // Base case: Stop when n reaches 0
        return;
    }
    System.out.println(n);
    printNumbers(n - 1); // Recursive call
}

public static void main(String[] args) {
    printNumbers(5);
}

```

Expected Output:

5
4
3
2
1



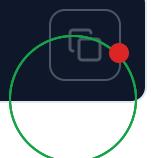
Explanation:

- If `n == 0`, the method stops (base case).
- Otherwise, it prints `n` and calls itself with `n - 1`, reducing the number each time until it reaches 0.

Factorial Using Recursion

Factorial of a number `n` is calculated as:

$$n! = n \times (n-1) \times (n-2) \times \dots \times 1$$



Using recursion, the factorial of **n** can be defined as:

```
factorial(n) = n × factorial(n-1)    (if n > 1)  
factorial(1) = 1      (base case)
```

Example: Factorial Calculation

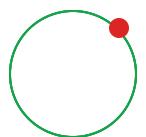
```
class FactorialExample {  
    // Recursive method to calculate factorial  
    static int factorial(int n) {  
        if (n == 1) { // Base case  
            return 1;  
        }  
        return n * factorial(n - 1); // Recursive call  
    }  
  
    public static void main(String[] args) {  
        int num = 5;  
        System.out.println("Factorial of " + num + " is: " + factorial(num));  
    }  
}
```

Expected Output:

```
Factorial of 5 is: 120
```

Explanation:

- When **n = 1**, the recursion stops and returns **1** (base case).
- Otherwise, the method calls itself with **n - 1**, multiplying **n** with the result of **factorial(n-1)**.
- The recursive calls unfold as:



```
factorial(5) → 5 × factorial(4)
factorial(4) → 4 × factorial(3)
factorial(3) → 3 × factorial(2)
factorial(2) → 2 × factorial(1)
factorial(1) → 1 (Base Case)
```

Fibonacci Series Using Recursion

The Fibonacci sequence is:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, ...
```

Each term is the sum of the two preceding numbers:

```
fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
```

Example: Fibonacci Series

```
class FibonacciExample {
    // Recursive method to find nth Fibonacci number
    static int fibonacci(int n) {
        if (n <= 1) { // Base case
            return n;
        }
        return fibonacci(n - 1) + fibonacci(n - 2); // Recursive call
    }

    public static void main(String[] args) {
        int terms = 7;
        System.out.print("Fibonacci Series: ");
        for (int i = 0; i < terms; i++) {
            System.out.print(fibonacci(i) + " ");
        }
    }
}
```

}

Expected Output:

Fibonacci Series: 0 1 1 2 3 5 8



Explanation:

- `fibonacci(0)` returns `0` and `fibonacci(1)` returns `1` (base case).
- `fibonacci(n)` calls `fibonacci(n-1)` and `fibonacci(n-2)`, summing their results.
- This process continues recursively until reaching the base case.

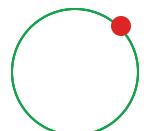
Recursion vs Iteration

Feature	Recursion	Iteration (Loop)
Speed	Slower (function call overhead)	Faster (no function calls)
Memory	Uses more memory (stack frames)	Uses less memory
Code Size	Short and easy to read	Longer and sometimes complex
Use Case	Problems like factorial, Fibonacci, tree traversal	Simple loops like printing numbers

Pros and Cons of Recursion

✓ Advantages:

- ✓ Simplifies complex problems.
- ✓ Reduces code length (compared to loops).



Java Tutorial – A Comprehensive Guide for Beginners

Introduction

Java is one of the most widely used and versatile programming languages in the world. From desktop applications to web services and mobile apps to enterprise systems, Java powers countless technologies and platforms.

This Java tutorial is designed for beginners who want to build a strong foundation in Java programming. Whether you're completely new to programming or transitioning from another language, this guide covers everything from core concepts to essential object-oriented principles and beyond.

If you're aiming to become a proficient Java developer, follow this structured path to master Java, step by step.

Prerequisites

Before you dive into Java, it's helpful (but not mandatory) to be familiar with:

- **Basic Computer Knowledge:** Understanding how software runs, what an IDE is, etc.
- **Logic Building:** Basic problem-solving and logical thinking.



- **Understanding of Programming Concepts** (optional): Knowing variables, loops, or functions from another language is a bonus.

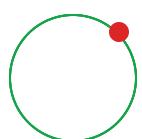
Master These Topics

Once you're ready, follow this structured list to learn Java from the ground up:

1. Introduction to Java
2. Java Fundamentals
3. Java Flow Control
4. Java Arrays
5. Java Methods
6. Java Object-Oriented Programming
7. Java Inner Classes
8. Static and Final Keywords
9. Java Exception Handling
10. Java Generics
11. Java Lambda Expressions
12. Java Collection Framework
13. Java Input/Output Streams
14. Java Reader/Writer

Introduction to Java

1. [Introduction to Java](#)
2. [Installing JDK on your computer](#)



[3. Java Comments](#)

Java Fundamentals

- [1. Java Variables and Literals](#)
- [2. Data Types in Java](#)
- [3. Operators in Java](#)
- [4. Java Basic Input and Output](#)
- [5. Java Expressions, Statements and Blocks](#)

Java Flow Control

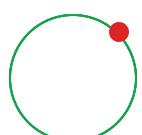
- [1. Java if else statement](#)
- [2. Java Ternary Operator](#)
- [3. Java For Loop](#)
- [4. Java while and do while loop](#)
- [5. Java continue and break statement](#)
- [6. Java Switch statement](#)

Java Arrays

- [1. Java Arrays](#)
- [2. Java Multidimensional Arrays](#)
- [3. Java Copy Arrays](#)

Java Methods

- [1. Methods in Java](#)





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



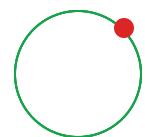
Java Object-Oriented Programming

1. [Principles of Object-Oriented Programming](#)
2. [Java Class and Objects](#)
3. [Java Constructor](#)
4. [Encapsulation and Data Hiding](#)
5. [Inheritance in Java](#)
6. [Constructors in Inheritance](#)
7. [this Vs super Keyword](#)
8. [Method Overriding](#)
9. [Dynamic Method Dispatch](#)
10. [Polymorphism using Overloading and Overriding](#)
11. [Abstract Classes](#)
12. [Interfaces in Java](#)

Java Inner Classes

1. [Inner Classes in Java](#)

Static and Final Keywords



1. Static Members and Static Blocks in Java
2. Final Members
3. Singleton Class

Java Exception Handling

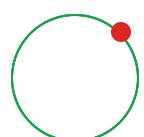
1. What are Exceptions
2. How to handle Exception
3. Try and Catch Block
4. Multiple and Nested Try Catch
5. Class Exception
6. Checked and Unchecked Exceptions
7. Throw Vs Throws
8. Finally Block
9. Try with Resources

Java Generics

1. Introduction to Generics
2. Defining Generic Class
3. Bounds on Generics
4. Java Generic Methods

Java Lambda Expressions

1. Introduction to Lambda Expressions
2. Parameters in Lambda Expressions



3. Capture in Lambda Expression

4. Method References

Java Collection Framework

1. Collections in Java

2. Understanding Collection Interface

3. Understanding List Interface in Java

4. Java ArrayList

5. Java LinkedList

6. Java Vector

7. Java Stack

8. Understanding Queue Interface in Java

9. Java PriorityQueue

10. Java ArrayDeque

11. Understanding Set Interface in Java

12. Java HashSet

13. Java LinkedHashSet

14. Java TreeSet

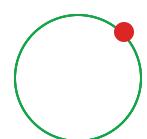
15. Understanding Map Interface in Java

16. Java HashMap

17. Java LinkedHashMap

18. Java TreeMap

19. Comparable Interface in Java



20. Comparator Interface in Java

Java Input/Output Streams

1. [What are Streams](#)
2. [InputStream and OutputStream Classes](#)
3. [FileInputStream and FileOutputStream](#)
4. [ByteArrayInputStream and ByteArrayOutputStream](#)
5. [ObjectInputStream and ObjectOutputStream](#)
6. [BufferedInputStream and BufferedOutputStream](#)
7. [Java PrintStream Class](#)

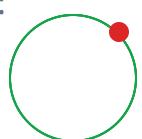
Java Reader/Writer

1. [Java File Class](#)
2. [Java Reader and Writer Class](#)
3. [InputStreamReader and OutputStreamWriter Class](#)
4. [FileReader and FileWriter Class](#)
5. [BufferedReader and BufferedWriter Class](#)
6. [StringReader and StringWriter Class](#)
7. [Java PrintWriter Class](#)

Key Features of Java

Java has remained popular for decades for good reason. Here's what makes it special:

- **Platform Independent:** Write once, run anywhere (thanks to the JVM).





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Principles of Object-Oriented Programming in Java

Introduction:

Object-Oriented Programming (OOP) is a programming paradigm that organizes code using **objects**—real-world representations that combine **data** and **behavior**. Java follows this model closely and encourages developers to build applications using this approach.

Before we explore the four core principles of OOP, it's essential to understand the fundamental building blocks: **classes** and **objects**.

What are Classes and Objects?

Class

A **class** is a blueprint or template that defines the structure and behavior of objects. It contains properties (fields or attributes) and methods (functions) that represent the characteristics and actions of a specific entity.

Object

An **object** is an instance of a class. It represents a specific real-world entity that has a state (data) and behavior (methods). Objects are created based on the structure defined by the class.



Real-World Analogy:

Think of a "Car" as a class. It defines properties like color and engine type, and behaviors like starting or braking. A specific red Honda Civic is an object created from that blueprint.

Understanding classes and objects forms the foundation of Object-Oriented Programming. Once you're comfortable with these, the four main principles of OOP become much easier to grasp.

Four Pillars of Object-Oriented Programming

OOP follows four fundamental principles:

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Let's explore each principle with real-world examples to understand their significance.

1. Encapsulation

Encapsulation is the process of bundling data (variables) and methods (functions) that operate on that data into a single unit called a class. It also restricts direct access to some of the object's components, which helps in protecting the integrity of the data.

Real-World Example:

Think of a **bank account**. You cannot directly access the account balance. Instead, you use **ATMs** or **online banking** to interact with your account through specific operations like **deposit** or **withdraw**.

- The **bank account details** (balance, account number) are hidden.
- The **ATM or banking system** provides a controlled interface to access and modify that data.

Encapsulation ensures that data remains secure and is only modified in controlled ways.

2. Abstraction

Abstraction focuses on **hiding the internal complexities** and only showing the essential features of an object. It simplifies interaction by exposing only what's necessary.

Real-World Example:

When you **drive a car**, you use the steering wheel, accelerator, and brake without needing to understand the internal workings of the engine or transmission.

- The **user interface** (steering, pedals) is exposed.
- The **internal mechanics** are hidden.

In programming, abstraction helps in reducing complexity and improving user experience by providing only the necessary functionality.

3. Inheritance

Inheritance is a mechanism where a **child class** derives properties and behaviors from a **parent class**. It promotes **code reusability** and a **hierarchical structure**.

Real-World Example:

Think of animals. There is a generic **Animal** category, and specific animals like **Dog**, **Cat**, or **Bird**.

- All animals share common traits (eating, sleeping, breathing).
- Each specific animal type extends these traits with its own behavior (e.g., a dog barks).

Inheritance helps avoid code duplication by allowing shared behavior to be defined once and reused across multiple classes.

4. Polymorphism

Polymorphism means **many forms**. It allows the same operation or object to behave differently in different contexts.

Real-World Example:

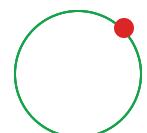
Consider a **smartphone's power button**:

- A **short press** turns the screen on or off.
- A **long press** brings up the power menu.
- A **double press** opens the camera.

In programming, polymorphism enables flexibility by allowing the same method or interface to perform different tasks depending on the object or context.

Conclusion

Object-Oriented Programming in Java is centered around the use of **classes** and **objects** to model real-world entities. Once the basic concepts are clear, the four principles—**Encapsulation, Abstraction, Inheritance, and Polymorphism**—offer a powerful way to design well-structured, reusable, and maintainable code.





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



Java Class and Objects

In Java, everything revolves around **classes and objects**. A class is a blueprint for creating objects, while an object is an instance of a class. Understanding these fundamental concepts is crucial for working with **Object-Oriented Programming (OOP)** in Java.

What is a Class?

A **class** is a template or blueprint that defines the attributes (fields) and behaviors (methods) of objects. It acts as a structure for creating multiple objects with similar properties.

Example:

Think of a **car manufacturing company**. Before making cars, the company designs a blueprint with specifications like color, engine type, and speed. This blueprint is like a **class** in Java.

In Java, a class is declared using the **class** keyword:

```
// Defining a class named Car
class Car {
    // Attributes (fields)
    String brand;
    int speed;

    // Method to display car details
    void displayCar() {
```



```
        System.out.println("Brand: " + brand);
        System.out.println("Speed: " + speed + " km/h");
    }
}
```

What is an Object?

An **object** is an instance of a class. When a class is defined, no memory is allocated until an object is created.

Example:

A real-world **car** (like a Tesla or BMW) is an **object** created from the blueprint (**class**). Each car object has its own unique attributes (color, model, speed).

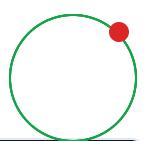
In Java, we create an object using the **new** keyword:

```
// Creating an object of the Car class
public class Main {
    public static void main(String[] args) {
        // Creating an object
        Car myCar = new Car();

        // Assigning values to object properties
        myCar.brand = "Tesla";
        myCar.speed = 200;

        // Calling the method
        myCar.displayCar();
    }
}
```

Output:





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



Java Constructor

In Java, a **constructor** is a special type of method that is used to initialize objects. It is automatically called when an object of a class is created. **Constructors help in setting initial values for object attributes.**

Characteristics of a Constructor

- A constructor **has the same name** as the class.
- It **does not have a return type**, not even **void**.
- It is **automatically invoked** when an object is created.
- Constructors **can have parameters** to initialize objects with different values.

Types of Constructors in Java

1. Default Constructor

A **default constructor** is a constructor **without parameters**. If no constructor is defined in a class, Java automatically provides a default constructor.

Example:



```
// Defining a class
class Car {
    // Constructor
    Car() {
        System.out.println("Car object is created!");
    }

    public static void main(String[] args) {
        // Creating an object of Car class
        Car myCar = new Car();
    }
}
```

Output:

```
Car object is created!
```



2. Parameterized Constructor

A parameterized constructor allows us to pass values while creating an object.

Example:

```
// Defining a class
class Car {
    String brand;
    int speed;

    // Parameterized Constructor
    Car(String b, int s) {
        brand = b;
        speed = s;
    }

    void display() {
        System.out.println("Brand: " + brand);
    }
}
```



```
        System.out.println("Speed: " + speed + " km/h");
    }

public static void main(String[] args) {
    // Creating objects with different values
    Car car1 = new Car("Tesla", 250);
    Car car2 = new Car("BMW", 220);

    // Displaying object details
    car1.display();
    car2.display();
}
```

Output:

```
Brand: Tesla
Speed: 250 km/h
Brand: BMW
Speed: 220 km/h
```

3. Copy Constructor

A **copy constructor** creates a new object by copying the values of an existing object.

Example:

```
class Car {
    String brand;
    int speed;

    // Parameterized Constructor
    Car(String b, int s) {
        brand = b;
        speed = s;
    }
}
```

```
// Copy Constructor
Car(Car obj) {
    brand = obj.brand;
    speed = obj.speed;
}

void display() {
    System.out.println("Brand: " + brand);
    System.out.println("Speed: " + speed + " km/h");
}

public static void main(String[] args) {
    Car car1 = new Car("Tesla", 250);
    Car car2 = new Car(car1); // Copying car1's values into car2

    car1.display();
    car2.display();
}
```

Output:

```
Brand: Tesla
Speed: 250 km/h
Brand: Tesla
Speed: 250 km/h
```

Constructor Overloading

Constructor overloading allows a class to have multiple constructors with different parameters.

Example:

```
class Car {
    String brand;
```

```

int speed;

// Default Constructor
Car() {
    brand = "Unknown";
    speed = 0;
}

// Parameterized Constructor
Car(String b, int s) {
    brand = b;
    speed = s;
}

void display() {
    System.out.println("Brand: " + brand);
    System.out.println("Speed: " + speed + " km/h");
}

public static void main(String[] args) {
    Car car1 = new Car(); // Calls default constructor
    Car car2 = new Car("Audi", 240); // Calls parameterized constructor

    car1.display();
    car2.display();
}

```

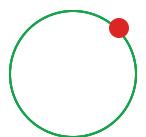
Output:

```

Brand: Unknown
Speed: 0 km/h
Brand: Audi
Speed: 240 km/h

```

Conclusion





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Encapsulation in Java

Encapsulation is one of the fundamental principles of Object-Oriented Programming (OOP). It is the concept of wrapping data (variables) and methods that operate on the data into a single unit called a class. It helps in restricting direct access to certain details of an object and only exposing necessary information through methods.

Why Encapsulation?

- **Data Hiding:** It prevents direct access to class fields, ensuring better control over data.
- **Modularity:** A class with encapsulated fields and methods makes it easier to manage code.
- **Maintainability:** Changing the internal implementation does not affect external code.
- **Security:** Restricts unauthorized access to sensitive data.

How to Implement Encapsulation?

Encapsulation in Java is implemented by:

- Declaring class variables as **private**.
- Providing **public** getter and setter methods to access and update the private field.



Example: Bank Account

Let's consider a bank account where we encapsulate the balance to ensure controlled access.

```
// BankAccount.java
class BankAccount {
    private double balance; // private variable

    // Constructor
    public BankAccount(double initialBalance) {
        if (initialBalance > 0) {
            this.balance = initialBalance;
        }
    }

    // Getter method to access balance
    public double getBalance() {
        return balance;
    }

    // Method to deposit money
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            System.out.println("Deposited: " + amount);
        } else {
            System.out.println("Invalid deposit amount.");
        }
    }

    // Method to withdraw money
    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
            System.out.println("Withdrawn: " + amount);
        } else {
            System.out.println("Insufficient funds or invalid amount.");
        }
    }
}
```

```
// Main class to test Encapsulation
public class Main {
    public static void main(String[] args) {
        // Create a bank account object
        BankAccount account = new BankAccount(5000);

        // Display initial balance
        System.out.println("Initial Balance: " + account.getBalance());

        // Deposit money
        account.deposit(2000);
        System.out.println("Updated Balance: " + account.getBalance());

        // Withdraw money
        account.withdraw(1000);
        System.out.println("Final Balance: " + account.getBalance());
    }
}
```

Expected Output

```
Initial Balance: 5000.0
Deposited: 2000.0
Updated Balance: 7000.0
Withdrawn: 1000.0
Final Balance: 6000.0
```

How Encapsulation Is Achieved in This Example

1. The `balance` field is declared `private`, so it can't be accessed directly from outside the class.
2. The `getBalance()`, `deposit()`, and `withdraw()` methods provide controlled access to the field.



 Pay Day Sale is Now Live!  Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



 Java Programming Handbook



What is Inheritance in Java?

Inheritance is one of the fundamental concepts of Object-Oriented Programming (OOP) that allows a class to inherit the properties and behaviors (fields and methods) of another class. It promotes **code reusability, modularity, and hierarchy in programming**.

Why Use Inheritance?

- **Code Reusability:** Reduces code duplication by allowing child classes to reuse existing code.
- **Modularity:** Enhances maintainability by keeping related functionalities within a structured hierarchy.
- **Extensibility:** Enables extending functionalities without modifying existing code.
- **Improved Readability:** Provides a clear relationship between classes, making the program easier to understand.

How Inheritance Works in Java?

In Java, inheritance is implemented using the **extends** keyword. The class that inherits another class is called a **child class (subclass)**, and the class being inherited is called a **parent class (superclass)**.



Example: Basic Inheritance

```
// Superclass
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}

// Subclass inheriting Animal class
class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks.");
    }
}

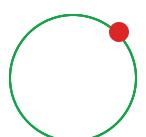
public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat(); // Inherited method
        dog.bark();
    }
}
```

Expected Output

```
This animal eats food.
Dog barks.
```

Types of Inheritance in Java

Java supports the following types of inheritance:



1. Single Inheritance

A subclass inherits from a single superclass.

```
class Vehicle {  
    void move() {  
        System.out.println("Vehicle is moving");  
    }  
}  
  
class Car extends Vehicle {  
    void speed() {  
        System.out.println("Car is speeding");  
    }  
}
```



2. Multilevel Inheritance

A subclass inherits from another subclass.

```
class Animal {  
    void sleep() {  
        System.out.println("Animal is sleeping");  
    }  
}  
  
class Mammal extends Animal {  
    void walk() {  
        System.out.println("Mammal walks");  
    }  
}  
  
class Dog extends Mammal {  
    void bark() {  
        System.out.println("Dog barks");  
    }  
}
```



3. Hierarchical Inheritance

Multiple subclasses inherit from a single superclass.

```
class Parent {  
    void show() {  
        System.out.println("Parent class");  
    }  
}  
  
class Child1 extends Parent {  
    void display() {  
        System.out.println("Child1 class");  
    }  
}  
  
class Child2 extends Parent {  
    void display() {  
        System.out.println("Child2 class");  
    }  
}
```

4. Hybrid Inheritance (Using Interfaces)

Java does **not support multiple inheritance** directly, but it can be achieved using **interfaces**.

```
interface A {  
    void methodA();  
}  
  
interface B {  
    void methodB();  
}
```

```
class C implements A, B {  
    public void methodA() {  
        System.out.println("Method A");  
    }  
    public void methodB() {  
        System.out.println("Method B");  
    }  
}
```

Why Multiple Inheritance is Not Supported in Java?

Multiple inheritance means a class inherits from more than one class. Java does not support it to avoid ambiguity and complexity.

Example: The Diamond Problem

```
class A {  
    void show() {  
        System.out.println("Class A");  
    }  
}  
  
class B extends A {  
    void show() {  
        System.out.println("Class B");  
    }  
}  
  
class C extends A {  
    void show() {  
        System.out.println("Class C");  
    }  
}  
  
// If Java allowed multiple inheritance  
// class D extends B, C {
```





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Constructors in Inheritance

In Java, when a class inherits another class, the constructor of the parent class is executed first, followed by the constructor of the child class. This ensures that the base class is properly initialized before the derived class adds its own functionality.

How Constructors Work in Inheritance

When an object of a derived class is created, the constructor of the base class is automatically called first. If there is no explicit call to the superclass constructor, Java automatically invokes the **default constructor** of the superclass.

Example 1: Default Constructor in Inheritance

```
class Parent {  
    Parent() {  
        System.out.println("Parent class constructor called");  
    }  
}  
  
class Child extends Parent {  
    Child() {  
        System.out.println("Child class constructor called");  
    }  
}
```



```
public class Main {  
    public static void main(String[] args) {  
        Child obj = new Child();  
    }  
}
```

Output:

```
Parent class constructor called  
Child class constructor called
```

Explanation:

- When we create an object of **Child**, the constructor of **Parent** is called first.
- After that, the constructor of **Child** is executed.

Using **super()** to Call Parameterized Constructor

By default, the Java compiler automatically calls the parent class's default constructor. However, if the parent class does not have a default constructor, we must explicitly call a parameterized constructor using **super()**.

Example 2: Using **super()** to Call Parameterized Constructor

```
class Parent {  
    Parent(String name) {  
        System.out.println("Parent constructor called. Name: " + name);  
    }  
}  
  
class Child extends Parent {  
    Child(String name) {  
        super(name);  
        System.out.println("Child constructor called. Name: " + name);  
    }  
}
```

```
super(name); // Calling the parent constructor
System.out.println("Child constructor called.");
}

}

public class Main {
    public static void main(String[] args) {
        Child obj = new Child("John");
    }
}
```

Output:

```
Parent constructor called. Name: John
Child constructor called.
```



Explanation:

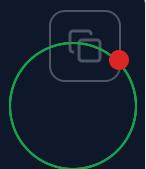
- The `super(name)` call explicitly invokes the constructor of the parent class.
- This ensures that the parent class is initialized properly before executing the child class constructor.

Constructor Chaining in Inheritance

Constructor chaining refers to the process where one constructor calls another constructor in the same or parent class using `this()` or `super()`.

Example 3: Constructor Chaining

```
class A {
    A() {
        System.out.println("Constructor of A");
```



```

    }
}

class B extends A {
    B() {
        super(); // Calls the constructor of A
        System.out.println("Constructor of B");
    }
}

class C extends B {
    C() {
        super(); // Calls the constructor of B
        System.out.println("Constructor of C");
    }
}

public class Main {
    public static void main(String[] args) {
        C obj = new C();
    }
}

```

Output:

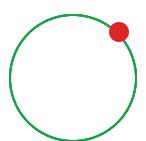
Constructor of A
 Constructor of B
 Constructor of C



Explanation:

- The `super()` calls ensure that constructors are called from the base class to the derived class in the correct order.

Conclusion





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



this vs super Keyword in Java

When working with Java classes, two important keywords, **this** and **super**, help in handling instance variables and inheritance. In this blog, we will understand these keywords with explanations and examples.

What is this Keyword?

The **this** keyword refers to the current instance of a class. It is mainly used to differentiate between instance variables and parameters with the same name, invoke constructors within the same class, and pass the current instance to a method.

Uses of this Keyword

- Referring to instance variables
- Calling another constructor in the same class
- Passing the current object as a parameter
- Returning the current instance from a method

Example: Using this to refer to instance variables

```
class Student {  
    String name;  
    int age;
```



```

// Constructor
Student(String name, int age) {
    this.name = name; // 'this' differentiates instance variable and para
    this.age = age;
}

void display() {
    System.out.println("Name: " + this.name + ", Age: " + this.age);
}
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student("John", 22);
        s1.display();
    }
}

```

Output:

Name: John, Age: 22



Example: Using this to call another constructor

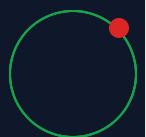
```

class Person {
    String name;
    int age;

    // Default constructor
    Person() {
        this("Unknown", 0); // Calls parameterized constructor
    }

    // Parameterized constructor
    Person(String name, int age) {

```



```
        this.name = name;
        this.age = age;
    }

    void display() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}

public class Main {
    public static void main(String[] args) {
        Person p1 = new Person();
        p1.display();
    }
}
```

Output:

Name: Unknown, Age: 0



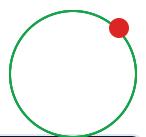
What is super Keyword?

The **super** keyword is used in inheritance to refer to the parent class. It is mainly used to access parent class members, invoke parent class methods, and call parent class constructors.

Uses of super Keyword

- Accessing parent class methods
- Calling the parent class constructor
- Accessing parent class variables

Example: Using super to call parent class methods



```
class Animal {  
    void makeSound() {  
        System.out.println("Animals make sound");  
    }  
}  
  
class Dog extends Animal {  
    void makeSound() {  
        super.makeSound(); // Calls the makeSound() of Animal class  
        System.out.println("Dogs bark");  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Dog d = new Dog();  
        d.makeSound();  
    }  
}
```

Output:

```
Animals make sound  
Dogs bark
```

Example: Using super to call parent class constructor

```
class Parent {  
    Parent() {  
        System.out.println("Parent class constructor");  
    }  
}  
  
class Child extends Parent {  
    Child() {  
        super(); // Calls the parent class constructor  
    }  
}
```

```

        System.out.println("Child class constructor");
    }
}

public class Main {
    public static void main(String[] args) {
        Child c = new Child();
    }
}

```

Output:

Parent class constructor
Child class constructor

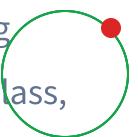


Difference Between this and super

Feature	this	super
Refers to	Current instance of the class	Parent class instance
Used for	Accessing instance variables, methods, and constructors	Accessing parent class members and constructors
Calls	Another constructor in the same class	Parent class constructor
Can be used in static methods?	No	No

Conclusion

In this blog, we learned about the **this** and **super** keywords in Java. The **this** keyword is used for referring to the current instance, differentiating instance variables, and calling constructors within the same class. The **super** keyword is used to refer to the parent class,





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Method Overriding in Java

In Java, **Method Overriding** is a feature that allows a subclass to provide a specific implementation of a method that is already defined in its parent class. The overridden method in the child class must have the same method signature as the one in the parent class.

Why Method Overriding?

Method Overriding is used to achieve **runtime polymorphism** and to provide a **specific implementation** for a method in a derived class that differs from its base class.

Key Rules of Method Overriding

- The method in the child class must have the **same name** as in the parent class.
- The method must have the **same parameters** as in the parent class.
- There must be an **inheritance relationship** (i.e., one class must be a subclass of another).
- The return type should be the same or **covariant** (subtype of the return type in the parent class).
- The access modifier **cannot be more restrictive** than the method in the parent class.
- Methods marked as **final**, **static**, or **private** cannot be overridden.

Example of Method Overriding



```
// Parent class
class Animal {
    void makeSound() {
        System.out.println("Animal makes a sound");
    }
}

// Child class
class Dog extends Animal {
    // Overriding the makeSound method
    @Override
    void makeSound() {
        System.out.println("Dog barks");
    }
}

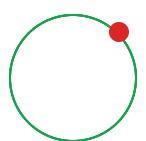
// Main class
public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Animal();
        myAnimal.makeSound(); // Calls method from Animal class

        Animal myDog = new Dog();
        myDog.makeSound(); // Calls overridden method from Dog class
    }
}
```

Output:

```
Animal makes a sound
Dog barks
```

Using super in Method Overriding



The **super** keyword can be used to call the overridden method of the parent class.

```
class Animal {  
    void makeSound() {  
        System.out.println("Animal makes a sound");  
    }  
}  
  
class Dog extends Animal {  
    @Override  
    void makeSound() {  
        super.makeSound(); // Calls method from parent class  
        System.out.println("Dog barks");  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Dog myDog = new Dog();  
        myDog.makeSound();  
    }  
}
```

Output:

```
Animal makes a sound  
Dog barks
```

Conclusion

In this blog, we learned about **Method Overriding** in Java, its importance in achieving **runtime polymorphism**, and the rules associated with it. We also explored examples demonstrating overriding and how to use **super** to call the parent class method. In the future blogs we will



 Pay Day Sale is Now Live!  Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



Dynamic Method Dispatch in Java

Introduction

Dynamic Method Dispatch, also known as **Runtime Polymorphism**, is one of the most powerful concepts in Java's object-oriented programming. It allows Java to determine which method to invoke at runtime rather than compile time. This feature is fundamental in achieving **method overriding** and is widely used in real-world applications.

Understanding Dynamic Method Dispatch

In Java, method calls are resolved dynamically at runtime using **Dynamic Method Dispatch**. This mechanism enables a **superclass reference variable** to refer to a **subclass object**, and Java determines which overridden method to execute based on the actual object type.

Key Points:

- It enables **runtime polymorphism**.
- Method invocation is determined by the object that the reference variable refers to (not the type of reference itself).
- It allows for **flexible and maintainable** code by supporting method overriding.



Example: Dynamic Method Dispatch

Let's take an example to understand Dynamic Method Dispatch in action:

```
// Parent class
class Animal {
    void makeSound() {
        System.out.println("Animals make different sounds");
    }
}

// Child class 1
class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Dog barks");
    }
}

// Child class 2
class Cat extends Animal {
    @Override
    void makeSound() {
        System.out.println("Cat meows");
    }
}

public class DynamicDispatchExample {
    public static void main(String[] args) {
        // Parent class reference pointing to child class object
        Animal myAnimal;

        // Assigning Dog object to Animal reference
        myAnimal = new Dog();
        myAnimal.makeSound(); // Output: Dog barks

        // Assigning Cat object to Animal reference
        myAnimal = new Cat();
        myAnimal.makeSound(); // Output: Cat meows
    }
}
```

```
    }  
}
```

Explanation of the Code:

1. The **Animal** class defines a method `makeSound()`, which is overridden by the **Dog** and **Cat** classes.
2. The reference variable **myAnimal** is of type **Animal**, but it holds objects of **Dog** and **Cat** at different times.
3. When `makeSound()` is called, Java determines the correct method implementation at runtime, based on the actual object type.

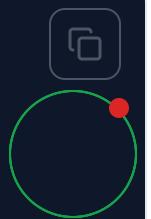
Why Use Dynamic Method Dispatch?

- **Achieves Runtime Polymorphism:** Allows the program to be more flexible and scalable.
- **Enhances Code Reusability:** A superclass reference can be used for multiple subclass objects.
- **Improves Maintainability:** Reduces dependencies between different parts of the code.

Real-World Use Case

Imagine you are developing a **payment system** where different payment methods (Credit Card, PayPal, UPI) should have their own processing logic. Using **Dynamic Method Dispatch**, you can achieve this as follows:

```
// Parent class  
class Payment {  
    void processPayment() {  
        System.out.println("Processing generic payment");  
    }  
}
```



```

    }
}

// Child class 1
class CreditCardPayment extends Payment {
    @Override
    void processPayment() {
        System.out.println("Processing credit card payment");
    }
}

// Child class 2
class PayPalPayment extends Payment {
    @Override
    void processPayment() {
        System.out.println("Processing PayPal payment");
    }
}

public class PaymentSystem {
    public static void main(String[] args) {
        Payment payment;

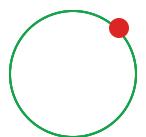
        // Process Credit Card Payment
        payment = new CreditCardPayment();
        payment.processPayment(); // Output: Processing credit card payment

        // Process PayPal Payment
        payment = new PayPalPayment();
        payment.processPayment(); // Output: Processing PayPal payment
    }
}

```

Important Notes

- Dynamic Method Dispatch only works with **method overriding**, not with method overloading.





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Polymorphism using Overloading and Overriding in Java

Polymorphism is one of the fundamental principles of Object-Oriented Programming (OOP). It allows a single interface to represent different types of actions. Java supports two types of polymorphism:

- **Compile-time polymorphism (Method Overloading)**
- **Runtime polymorphism (Method Overriding)**

We have already covered **Method Overriding** and **Dynamic Method Dispatch** in detail. In this blog, we will explore both **Method Overloading** and how it works alongside Overriding in achieving polymorphism in Java.

Method Overloading (Compile-Time Polymorphism)

Method overloading in Java allows multiple methods to have the same name but different parameters. The compiler determines which method to call based on the method signature (number and type of parameters).

Example of Method Overloading

```
class MathOperations {  
    // Method to add two integers
```



```

        int add(int a, int b) {
            return a + b;
        }

        // Method to add three integers
        int add(int a, int b, int c) {
            return a + b + c;
        }

        // Method to add two double numbers
        double add(double a, double b) {
            return a + b;
        }
    }

public class OverloadingExample {
    public static void main(String[] args) {
        MathOperations obj = new MathOperations();

        System.out.println("Sum of two integers: " + obj.add(5, 10));
        System.out.println("Sum of three integers: " + obj.add(5, 10, 15));
        System.out.println("Sum of two double values: " + obj.add(5.5, 2.2));
    }
}

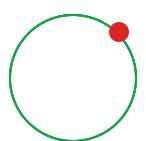
```

Expected Output:

Sum of two integers: 15
 Sum of three integers: 30
 Sum of two double values: 7.7

Key Points about Method Overloading:

- Overloaded methods must have different parameter lists.
- The return type can be different, but it does not determine overloading.



- It improves code readability and reusability.
- It occurs at compile time, making it **compile-time polymorphism**.

Method Overriding (Runtime Polymorphism)

Method overriding occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. The method in the subclass must have the same signature as in the superclass.

Example of Method Overriding

```
class Animal {  
    void makeSound() {  
        System.out.println("Animal makes a sound");  
    }  
}  
  
class Dog extends Animal {  
    @Override  
    void makeSound() {  
        System.out.println("Dog barks");  
    }  
}  
  
public class OverridingExample {  
    public static void main(String[] args) {  
        Animal myAnimal = new Dog(); // Upcasting  
        myAnimal.makeSound(); // Calls the overridden method in Dog class  
    }  
}
```

Expected Output:





Key Points about Method Overriding:

- The method name and parameters must match exactly with the superclass method.
- The method in the subclass should have the same return type or a subtype.
- It occurs at runtime, making it **runtime polymorphism**.
- The `@Override` annotation is used to indicate an overridden method.

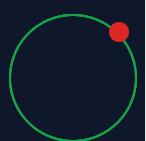
Polymorphism using Overloading and Overriding

Both overloading and overriding contribute to achieving polymorphism in Java:

- **Method Overloading** provides flexibility by allowing multiple methods with the same name but different parameters within the same class.
- **Method Overriding** allows dynamic method invocation, enabling different behaviors based on the object type at runtime.

Example Demonstrating Both Overloading and Overriding

```
class Shape {  
    // Overloaded method (Compile-time polymorphism)  
    void draw() {  
        System.out.println("Drawing a shape");  
    }  
  
    void draw(String shapeType) {  
        System.out.println("Drawing a " + shapeType);  
    }  
}  
  
class Circle extends Shape {
```



```

// Overridden method (Runtime polymorphism)
@Override
void draw() {
    System.out.println("Drawing a Circle");
}

public class PolymorphismExample {
    public static void main(String[] args) {
        Shape shape1 = new Shape();
        shape1.draw(); // Calls draw() from Shape
        shape1.draw("Square"); // Calls overloaded draw() from Shape

        Shape shape2 = new Circle(); // Upcasting
        shape2.draw(); // Calls overridden draw() from Circle
    }
}

```

Expected Output:

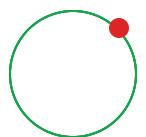
Drawing a shape
Drawing a Square
Drawing a Circle



Conclusion

In this blog, we learned about **Polymorphism in Java using Overloading and Overriding**. We explored:

- **Method Overloading**, which occurs at compile-time and enables multiple methods with the same name but different parameters.
- **Method Overriding**, which allows runtime polymorphism by enabling a subclass to modify the behavior of a superclass method.
- How both concepts together enable flexible and dynamic method execution.





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Abstract Classes in Java

Introduction

In Java, sometimes we want to create a class that should not be instantiated directly but instead should serve as a blueprint for other classes. This is where **abstract classes** come into play!

An **abstract class** is a class that **cannot be instantiated** and may contain **abstract methods** (methods without a body) that must be implemented by its subclasses.

Think of an abstract class as a **template**—it provides some functionality but leaves specific details for subclasses to define.

Understanding Abstract Classes with a Real-World Example

Imagine you are developing a program for different types of vehicles. All vehicles share some common properties (like speed, fuel type) but have different ways of moving (cars drive, planes fly, boats sail).

Here, you can create an **abstract class** `Vehicle` that has general properties and methods but leaves the implementation of `move()` to specific vehicle types.





```
// Abstract class
abstract class Vehicle {
    int speed;
    String fuelType;

    // Constructor
    Vehicle(int speed, String fuelType) {
        this.speed = speed;
        this.fuelType = fuelType;
    }

    // Abstract method (to be implemented by subclasses)
    abstract void move();

    // Concrete method (common functionality for all vehicles)
    void showInfo() {
        System.out.println("Speed: " + speed + " km/h");
        System.out.println("Fuel Type: " + fuelType);
    }
}

// Subclass Car
class Car extends Vehicle {
    Car(int speed, String fuelType) {
        super(speed, fuelType);
    }

    @Override
    void move() {
        System.out.println("Car moves on roads.");
    }
}

// Subclass Airplane
class Airplane extends Vehicle {
    Airplane(int speed, String fuelType) {
        super(speed, fuelType);
    }

    @Override
```



```

    void move() {
        System.out.println("Airplane flies in the sky.");
    }
}

// Main class to test
public class AbstractExample {
    public static void main(String[] args) {
        Vehicle car = new Car(120, "Petrol");
        car.showInfo();
        car.move();

        System.out.println();

        Vehicle airplane = new Airplane(800, "Jet Fuel");
        airplane.showInfo();
        airplane.move();
    }
}

```

Output:

Speed: 120 km/h
 Fuel Type: Petrol
 Car moves on roads.

Speed: 800 km/h
 Fuel Type: Jet Fuel
 Airplane flies in the sky.

Key Features of Abstract Classes

- ✓ **Cannot be instantiated** – You cannot create an object of an abstract class.
- ✓ **Can have abstract methods** – Methods without implementation that subclasses must define.
- ✓ **Can have concrete methods** – Fully implemented methods that all subclasses inherit.
- ✓ **Can have**

constructors – Used to initialize fields, just like a normal class. ✓ **Can contain variables** – Both instance and static variables are allowed.

Do's and Don'ts of Abstract Classes (with Examples)

✓ Do's

1. Use abstract classes when you want to enforce common functionality

If multiple classes share some behavior but should also have their unique implementations, an abstract class is a great choice.

```
abstract class Animal {  
    abstract void makeSound();  
  
    void sleep() {  
        System.out.println("Sleeping...");  
    }  
}  
  
class Dog extends Animal {  
    @Override  
    void makeSound() {  
        System.out.println("Barks");  
    }  
}
```

2. Use abstract classes when you need a constructor for common properties

Unlike interfaces, abstract classes **can have constructors** to initialize common attributes.

```
abstract class Employee {  
    String name;  
    int salary;  
  
    Employee(String name, int salary) {
```

```
        this.name = name;
        this.salary = salary;
    }
}
```

✖ Don'ts

1. Don't instantiate an abstract class

Abstract classes are meant to be extended, **not instantiated**.

✖ Incorrect:

```
abstract class Shape {}
public class Test {
    public static void main(String[] args) {
        Shape s = new Shape(); // ERROR: Cannot instantiate abstract class
    }
}
```



2. Don't declare an abstract method in a class that can be instantiated

If a class has an **abstract method**, the class itself must be abstract.

✖ Incorrect:

```
class Vehicle { // Should be abstract
    abstract void move(); // ERROR: Must declare class as abstract
}
```



✓ Correct:



```
abstract class Vehicle {  
    abstract void move();  
}
```

3. Don't forget to override abstract methods in subclasses

If a subclass does not implement an abstract method, it must also be declared abstract.

 Incorrect:

```
abstract class Animal {  
    abstract void sound();  
}  
class Dog extends Animal {} // ERROR: Must implement sound() method
```

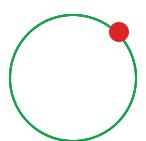
 Correct:

```
class Dog extends Animal {  
    @Override  
    void sound() {  
        System.out.println("Bark");  
    }  
}
```

Conclusion

In this blog, we learned about **Abstract Classes in Java**. We covered:

- What abstract classes are and why they are useful.
- A real-world example demonstrating how to use them.
- Key do's and don'ts with explanations and examples.





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Interfaces in Java

In Java, an **interface** is a blueprint of a class that contains only **abstract methods** (methods without a body) and **constants** (variables declared as **public**, **static**, and **final** by default). Interfaces help achieve **abstraction** and **multiple inheritance**, making Java programs more modular and maintainable.

Why Use Interfaces?

- **Achieve Abstraction:** Interfaces allow us to define methods without implementing them, leaving the implementation to child classes.
- **Support Multiple Inheritance:** Unlike classes, Java allows a class to implement multiple interfaces, overcoming the limitations of single inheritance.
- **Ensure Loose Coupling:** Interfaces separate the definition of functionality from implementation, making code more flexible and scalable.

Defining and Implementing an Interface

Example: Defining an Interface

```
interface Animal {  
    void makeSound(); // Abstract method (no body)  
}
```



Example: Implementing an Interface in a Class

```
class Dog implements Animal {  
    @Override  
    public void makeSound() {  
        System.out.println("Dog barks");  
    }  
}  
  
public class InterfaceExample {  
    public static void main(String[] args) {  
        Animal myDog = new Dog(); // Upcasting  
        myDog.makeSound();  
    }  
}
```

Output:

```
Dog barks
```

Here, **Dog** implements the **Animal** interface by providing a concrete definition for the **makeSound()** method.

Multiple Interfaces in Java

A class can implement multiple interfaces, which is not possible with regular class inheritance.

Example: Implementing Multiple Interfaces

```
interface Flyable {  
    void fly();  
}  
  
interface Swimmable {
```

```
void swim();  
}  
  
class Duck implements Flyable, Swimmable {  
    @Override  
    public void fly() {  
        System.out.println("Duck is flying");  
    }  
  
    @Override  
    public void swim() {  
        System.out.println("Duck is swimming");  
    }  
}  
  
public class MultipleInterfacesExample {  
    public static void main(String[] args) {  
        Duck myDuck = new Duck();  
        myDuck.fly();  
        myDuck.swim();  
    }  
}
```

Output:

```
Duck is flying  
Duck is swimming
```

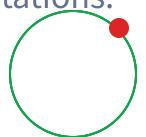


This example shows how a class (**Duck**) can implement multiple interfaces (**Flyable** and **Swimmable**) and provide implementations for both.

Default and Static Methods in Interfaces

Since Java 8, interfaces can have **default** and **static** methods with concrete implementations.

Example of Default and Static Methods



```
interface Vehicle {  
    void start(); // Abstract method  
  
    // Default method with a body  
    default void stop() {  
        System.out.println("Vehicle is stopping");  
    }  
  
    // Static method with a body  
    static void maintenance() {  
        System.out.println("Vehicle requires maintenance");  
    }  
}  
  
class Car implements Vehicle {  
    @Override  
    public void start() {  
        System.out.println("Car is starting");  
    }  
}  
  
public class InterfaceMethodsExample {  
    public static void main(String[] args) {  
        Car myCar = new Car();  
        myCar.start(); // Calls overridden method  
        myCar.stop(); // Calls default method  
        Vehicle.maintenance(); // Calls static method  
    }  
}
```

Output:

```
Car is starting  
Vehicle is stopping  
Vehicle requires maintenance
```

- **Default methods** allow interfaces to have some behavior without breaking existing implementations.
- **Static methods** belong to the interface itself and can be called without an instance.

Abstract Classes vs Interfaces

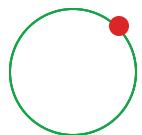
Feature	Abstract Class	Interface
Can have constructors?	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> No
Can have abstract methods?	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> Yes
Can have concrete methods?	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> (Since Java 8)
Can have instance variables?	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> No (Only constants allowed)
Supports multiple inheritance?	<input checked="" type="checkbox"/> No	<input checked="" type="checkbox"/> Yes

Do's and Don'ts of Interfaces

Do's:

- Use interfaces for abstraction:** If you only need to define behavior without implementing it, use an interface instead of an abstract class.
- Use interfaces for multiple inheritance:** If a class needs behavior from multiple sources, interfaces help avoid the **diamond problem** found in multiple inheritance.
- Use default methods to provide optional implementations:** If an interface needs to add new behavior without breaking existing classes, default methods help maintain backward compatibility.

Don'ts:



Java Tutorial – A Comprehensive Guide for Beginners

Introduction

Java is one of the most widely used and versatile programming languages in the world. From desktop applications to web services and mobile apps to enterprise systems, Java powers countless technologies and platforms.

This Java tutorial is designed for beginners who want to build a strong foundation in Java programming. Whether you're completely new to programming or transitioning from another language, this guide covers everything from core concepts to essential object-oriented principles and beyond.

If you're aiming to become a proficient Java developer, follow this structured path to master Java, step by step.

Prerequisites

Before you dive into Java, it's helpful (but not mandatory) to be familiar with:

- **Basic Computer Knowledge:** Understanding how software runs, what an IDE is, etc.
- **Logic Building:** Basic problem-solving and logical thinking.



- **Understanding of Programming Concepts** (optional): Knowing variables, loops, or functions from another language is a bonus.

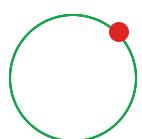
Master These Topics

Once you're ready, follow this structured list to learn Java from the ground up:

1. Introduction to Java
2. Java Fundamentals
3. Java Flow Control
4. Java Arrays
5. Java Methods
6. Java Object-Oriented Programming
7. Java Inner Classes
8. Static and Final Keywords
9. Java Exception Handling
10. Java Generics
11. Java Lambda Expressions
12. Java Collection Framework
13. Java Input/Output Streams
14. Java Reader/Writer

Introduction to Java

1. [Introduction to Java](#)
2. [Installing JDK on your computer](#)



[3. Java Comments](#)

Java Fundamentals

- [1. Java Variables and Literals](#)
- [2. Data Types in Java](#)
- [3. Operators in Java](#)
- [4. Java Basic Input and Output](#)
- [5. Java Expressions, Statements and Blocks](#)

Java Flow Control

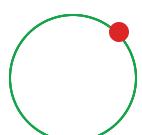
- [1. Java if else statement](#)
- [2. Java Ternary Operator](#)
- [3. Java For Loop](#)
- [4. Java while and do while loop](#)
- [5. Java continue and break statement](#)
- [6. Java Switch statement](#)

Java Arrays

- [1. Java Arrays](#)
- [2. Java Multidimensional Arrays](#)
- [3. Java Copy Arrays](#)

Java Methods

- [1. Methods in Java](#)



2. Parameter Passing in Java
3. Method Overloading
4. Variable Arguments
5. Java CommandLine Arguments
6. Java Recursive Method

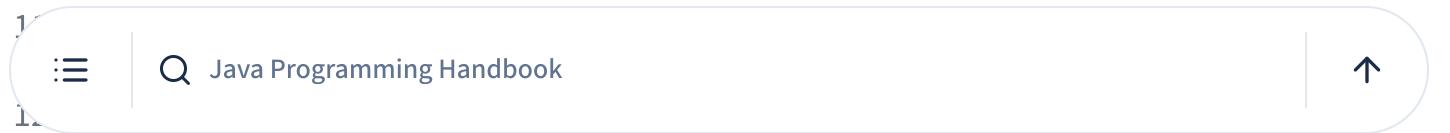
Java Object-Oriented Programming

1. Principles of Object-Oriented Programming
2. Java Class and Objects
3. Java Constructor
4. Encapsulation and Data Hiding
5. Inheritance in Java
6. Constructors in Inheritance
7. this Vs super Keyword

Method Overriding



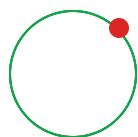
Pay Day Sale is Now Live! Use the Coupon PAYDAY and Get Maximum Discount [Enrol Now!](#).



Java Inner Classes

1. Inner Classes in Java

Static and Final Keywords



1. Static Members and Static Blocks in Java
2. Final Members
3. Singleton Class

Java Exception Handling

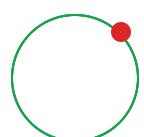
1. What are Exceptions
2. How to handle Exception
3. Try and Catch Block
4. Multiple and Nested Try Catch
5. Class Exception
6. Checked and Unchecked Exceptions
7. Throw Vs Throws
8. Finally Block
9. Try with Resources

Java Generics

1. Introduction to Generics
2. Defining Generic Class
3. Bounds on Generics
4. Java Generic Methods

Java Lambda Expressions

1. Introduction to Lambda Expressions
2. Parameters in Lambda Expressions



3. Capture in Lambda Expression

4. Method References

Java Collection Framework

1. Collections in Java

2. Understanding Collection Interface

3. Understanding List Interface in Java

4. Java ArrayList

5. Java LinkedList

6. Java Vector

7. Java Stack

8. Understanding Queue Interface in Java

9. Java PriorityQueue

10. Java ArrayDeque

11. Understanding Set Interface in Java

12. Java HashSet

13. Java LinkedHashSet

14. Java TreeSet

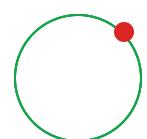
15. Understanding Map Interface in Java

16. Java HashMap

17. Java LinkedHashMap

18. Java TreeMap

19. Comparable Interface in Java



20. Comparator Interface in Java

Java Input/Output Streams

1. [What are Streams](#)
2. [InputStream and OutputStream Classes](#)
3. [FileInputStream and FileOutputStream](#)
4. [ByteArrayInputStream and ByteArrayOutputStream](#)
5. [ObjectInputStream and ObjectOutputStream](#)
6. [BufferedInputStream and BufferedOutputStream](#)
7. [Java PrintStream Class](#)

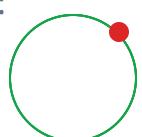
Java Reader/Writer

1. [Java File Class](#)
2. [Java Reader and Writer Class](#)
3. [InputStreamReader and OutputStreamWriter Class](#)
4. [FileReader and FileWriter Class](#)
5. [BufferedReader and BufferedWriter Class](#)
6. [StringReader and StringWriter Class](#)
7. [Java PrintWriter Class](#)

Key Features of Java

Java has remained popular for decades for good reason. Here's what makes it special:

- **Platform Independent:** Write once, run anywhere (thanks to the JVM).





 Pay Day Sale is Now Live!  Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



 Java Programming Handbook



Inner Classes in Java

Introduction

In Java, **Inner Classes** are classes that are defined within another class. They provide better encapsulation, logical grouping of related classes, and improved code organization. Understanding inner classes is crucial as they are often used in real-world scenarios like event handling, multithreading, and UI development.

In this blog, we will cover:

- What are inner classes?
- Why do we need inner classes?
- Types of inner classes with examples.
- When and where to use inner classes.

1. What are Inner Classes?

An **Inner Class** is a class that is declared **inside another class or interface**. The inner class has access to all members of its enclosing class, even private members.

Syntax:



```
class OuterClass {  
    class InnerClass {  
        // Inner class code  
    }  
}
```

2. Why Do We Need Inner Classes?

Inner classes are useful in several scenarios:

- **Encapsulation:** Helps in logically grouping classes that are only used by the enclosing class.
- **Code Readability:** Reduces clutter by keeping closely related logic together.
- **Event Handling:** Commonly used in GUI applications and event-driven programming (e.g., in Swing or JavaFX).
- **Accessing Private Members:** Inner classes can access private members of the outer class, making them useful in scenarios requiring close interaction between classes.

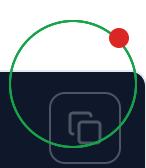
3. Types of Inner Classes in Java

Java provides four types of inner classes:

3.1. Member Inner Class

A **Member Inner Class** is a non-static class that is defined inside another class. It can access all members (even private) of the outer class.

Example:



```

class Outer {
    private String message = "Hello from Outer class!";

    class Inner {
        void showMessage() {
            System.out.println(message); // Accessing private member
        }
    }

    public static void main(String[] args) {
        Outer outer = new Outer();
        Outer.Inner inner = outer.new Inner(); // Creating instance of Inner
        inner.showMessage();
    }
}

```

Output:

Hello from Outer class!



3.2. Static Nested Class

Unlike member inner classes, a **Static Nested Class** is declared with the **static** keyword and cannot access non-static members of the outer class directly.

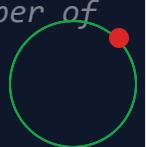
Example:

```

class Outer {
    static String staticMessage = "Hello from Static Outer Class!";

    static class StaticNested {
        void display() {
            System.out.println(staticMessage); // Accessing static member of
        }
    }
}

```



```
public static void main(String[] args) {
    Outer.StaticNested nested = new Outer.StaticNested(); // No need for
    nested.display();
}
}
```

Output:

```
Hello from Static Outer Class!
```



3.3. Local Inner Class

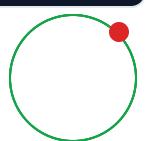
A **Local Inner Class** is defined within a method and can only be used inside that method.

Example:

```
class Outer {
    void outerMethod() {
        class LocalInner {
            void display() {
                System.out.println("Inside Local Inner Class");
            }
        }
        LocalInner local = new LocalInner();
        local.display();
    }

    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.outerMethod();
    }
}
```

Output:





3.4. Anonymous Inner Class

An **Anonymous Inner Class** is a class that **does not have a name** and is used when we need to override a method of a class or an interface inline.

Example:

```
abstract class Animal {  
    abstract void sound();  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Animal cat = new Animal() { // Anonymous Inner Class  
            void sound() {  
                System.out.println("Meow Meow");  
            }  
        };  
        cat.sound();  
    }  
}
```



Output:

```
Meow Meow
```



4. When and Where to Use Inner Classes?

Now that we have explored different types of inner classes, let's see where they are useful:



Java Tutorial – A Comprehensive Guide for Beginners

Introduction

Java is one of the most widely used and versatile programming languages in the world. From desktop applications to web services and mobile apps to enterprise systems, Java powers countless technologies and platforms.

This Java tutorial is designed for beginners who want to build a strong foundation in Java programming. Whether you're completely new to programming or transitioning from another language, this guide covers everything from core concepts to essential object-oriented principles and beyond.

If you're aiming to become a proficient Java developer, follow this structured path to master Java, step by step.

Prerequisites

Before you dive into Java, it's helpful (but not mandatory) to be familiar with:

- **Basic Computer Knowledge:** Understanding how software runs, what an IDE is, etc.
- **Logic Building:** Basic problem-solving and logical thinking.



- **Understanding of Programming Concepts** (optional): Knowing variables, loops, or functions from another language is a bonus.

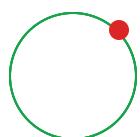
Master These Topics

Once you're ready, follow this structured list to learn Java from the ground up:

1. Introduction to Java
2. Java Fundamentals
3. Java Flow Control
4. Java Arrays
5. Java Methods
6. Java Object-Oriented Programming
7. Java Inner Classes
8. Static and Final Keywords
9. Java Exception Handling
10. Java Generics
11. Java Lambda Expressions
12. Java Collection Framework
13. Java Input/Output Streams
14. Java Reader/Writer

Introduction to Java

1. [Introduction to Java](#)
2. [Installing JDK on your computer](#)



[3. Java Comments](#)

Java Fundamentals

- [1. Java Variables and Literals](#)
- [2. Data Types in Java](#)
- [3. Operators in Java](#)
- [4. Java Basic Input and Output](#)
- [5. Java Expressions, Statements and Blocks](#)

Java Flow Control

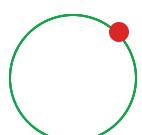
- [1. Java if else statement](#)
- [2. Java Ternary Operator](#)
- [3. Java For Loop](#)
- [4. Java while and do while loop](#)
- [5. Java continue and break statement](#)
- [6. Java Switch statement](#)

Java Arrays

- [1. Java Arrays](#)
- [2. Java Multidimensional Arrays](#)
- [3. Java Copy Arrays](#)

Java Methods

- [1. Methods in Java](#)



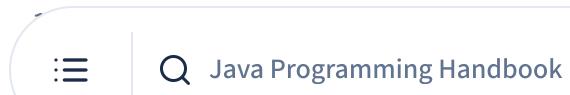
2. Parameter Passing in Java
3. Method Overloading
4. Variable Arguments
5. Java CommandLine Arguments
6. Java Recursive Method

Java Object-Oriented Programming

1. Principles of Object-Oriented Programming
2. Java Class and Objects
3. Java Constructor
4. Encapsulation and Data Hiding
5. Inheritance in Java
6. Constructors in Inheritance
7. this Vs super Keyword
8. Method Overriding
9. Dynamic Method Dispatch
10. Polymorphism using Overloading and Overriding
11. Abstract Classes



Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Static and Final Keywords

1. Static Members and Static Blocks in Java
2. Final Members
3. Singleton Class

Java Exception Handling

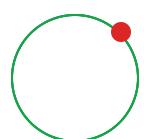
1. What are Exceptions
2. How to handle Exception
3. Try and Catch Block
4. Multiple and Nested Try Catch
5. Class Exception
6. Checked and Unchecked Exceptions
7. Throw Vs Throws
8. Finally Block
9. Try with Resources

Java Generics

1. Introduction to Generics
2. Defining Generic Class
3. Bounds on Generics
4. Java Generic Methods

Java Lambda Expressions

1. Introduction to Lambda Expressions
2. Parameters in Lambda Expressions



3. Capture in Lambda Expression

4. Method References

Java Collection Framework

1. Collections in Java

2. Understanding Collection Interface

3. Understanding List Interface in Java

4. Java ArrayList

5. Java LinkedList

6. Java Vector

7. Java Stack

8. Understanding Queue Interface in Java

9. Java PriorityQueue

10. Java ArrayDeque

11. Understanding Set Interface in Java

12. Java HashSet

13. Java LinkedHashSet

14. Java TreeSet

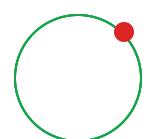
15. Understanding Map Interface in Java

16. Java HashMap

17. Java LinkedHashMap

18. Java TreeMap

19. Comparable Interface in Java



20. Comparator Interface in Java

Java Input/Output Streams

1. [What are Streams](#)
2. [InputStream and OutputStream Classes](#)
3. [FileInputStream and FileOutputStream](#)
4. [ByteArrayInputStream and ByteArrayOutputStream](#)
5. [ObjectInputStream and ObjectOutputStream](#)
6. [BufferedInputStream and BufferedOutputStream](#)
7. [Java PrintStream Class](#)

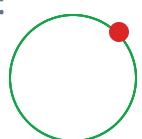
Java Reader/Writer

1. [Java File Class](#)
2. [Java Reader and Writer Class](#)
3. [InputStreamReader and OutputStreamWriter Class](#)
4. [FileReader and FileWriter Class](#)
5. [BufferedReader and BufferedWriter Class](#)
6. [StringReader and StringWriter Class](#)
7. [Java PrintWriter Class](#)

Key Features of Java

Java has remained popular for decades for good reason. Here's what makes it special:

- **Platform Independent:** Write once, run anywhere (thanks to the JVM).





 Pay Day Sale is Now Live!  Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



 Java Programming Handbook



Static Members and Static Blocks in Java

Introduction

In Java, the **static** keyword is used for memory management and allows defining class-level variables and methods. The **static** block, also known as a **static initializer**, is executed when the class is loaded into memory. This blog will help beginners understand **static members** and **static blocks** in Java with real-world examples.

Topics Covered:

- What are Static Members?
- What are Static Blocks?
- Examples and Use Cases.
- Key Differences Between Static Members and Instance Members.
- When to Use Static Members and Static Blocks?

1. What are Static Members in Java?

Static Members (variables and methods) belong to the class rather than an instance of the class. They are shared among all instances of the class and are accessed using the class



Characteristics of Static Members:

- Shared across all instances of the class.
- Stored in the class area of memory.
- Can be accessed directly using the class name without creating an instance.
- Cannot use instance variables or methods directly since they do not belong to an instance.

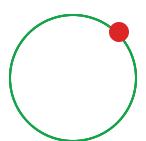
Example of Static Variables and Methods:

```
class Company {  
    static String companyName = "TechCorp"; // Static Variable  
  
    static void displayCompany() { // Static Method  
        System.out.println("Company Name: " + companyName);  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        // Accessing static members without creating an object  
        Company.displayCompany();  
        System.out.println("Company: " + Company.companyName);  
    }  
}
```

Output:

```
Company Name: TechCorp  
Company: TechCorp
```

2. What is a Static Block in Java?



A **Static Block** is a block of code enclosed in `{}` and preceded by the `static` keyword. It is executed **only once when the class is loaded** into memory.

Characteristics of Static Blocks:

- Executed automatically when the class is loaded, even before the `main()` method.
- Used for initializing static variables before any objects are created.
- Can have multiple static blocks, which execute in order.

Example of a Static Block:

```
class Database {  
    static String connectionURL;  
  
    // Static Block  
    static {  
        connectionURL = "jdbc:mysql://localhost:3306/mydb";  
        System.out.println("Static Block Executed: Database Connection Initialized");  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        System.out.println("Database URL: " + Database.connectionURL);  
    }  
}
```

Output:

```
Static Block Executed: Database Connection Initialized  
Database URL: jdbc:mysql://localhost:3306/mydb
```

3. Key Differences Between Static Members and Instance Members

Feature	Static Members	Instance Members
Belongs to	Class	Individual objects
Memory Storage	Class area (Method area)	Heap memory
Access	Can be accessed using class name	Requires object creation
Instance Access	Cannot access instance members directly	Can access both static and instance members
Initialization	Initialized once when the class is loaded	Initialized when objects are created

4. When to Use Static Members and Static Blocks?

Use Case	Static Members	Static Blocks
Shared data among all objects (e.g., constants)	<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No
Utility methods (e.g., <code>Math.pow()</code>)	<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No
Database connections (initialization)	<input type="checkbox"/> No	<input checked="" type="checkbox"/> Yes
Complex one-time initialization logic	<input type="checkbox"/> No	<input checked="" type="checkbox"/> Yes
Executing code before <code>main()</code> method	<input type="checkbox"/> No	<input checked="" type="checkbox"/> Yes

Conclusion

In this blog, we covered:

- **Static members (variables and methods)** that belong to the class and are shared  among all instances.



Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



Final Members in Java

Introduction

In Java, the **final** keyword is used to impose restrictions on variables, methods, and classes. Once a member is declared **final**, it cannot be modified, overridden, or extended, depending on its usage. This blog will guide beginners through the concept of **final members**, explain their significance, and provide real-world examples.

Topics Covered:

- What are Final Members in Java?
- Final Variables
- Final Methods
- Final Classes
- Differences Between Final, Static, and Constant Variables
- When to Use Final Members?

1. What are Final Members in Java?

The **final** keyword in Java is a modifier that can be applied to **variables**, **methods**, and **classes**. Its primary purpose is to **prevent modifications** and ensure stability in the code.



Characteristics of Final Members:

- **Final variables** cannot be reassigned after initialization.
- **Final methods** cannot be overridden in subclasses.
- **Final classes** cannot be extended by other classes.
- Helps maintain **immutability** and **security** in Java applications.

2. Final Variables in Java

A **final variable** is a constant whose value cannot be changed after initialization. It can be initialized at the time of declaration or inside a constructor.

Example of Final Variable:

```
class Configuration {  
    final int MAX_USERS = 100; // Final Variable  
  
    void displayLimit() {  
        System.out.println("Maximum users allowed: " + MAX_USERS);  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Configuration config = new Configuration();  
        config.displayLimit();  
        // config.MAX_USERS = 200; // This will cause a compilation error  
    }  
}
```

Output:





Example of Final Variable Initialized in Constructor:

```
class Person {  
    final String name;  
  
    // Constructor to initialize final variable  
    Person(String name) {  
        this.name = name;  
    }  
  
    void display() {  
        System.out.println("Name: " + name);  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Person p = new Person("Alice");  
        p.display();  
        // p.name = "Bob"; // Compilation error  
    }  
}
```



3. Final Methods in Java

A **final method** prevents subclasses from overriding the method, ensuring that the original implementation remains unchanged.

Example of a Final Method:

```
class Parent {  
    final void showMessage() {
```



```
        System.out.println("This message cannot be overridden.");
    }
}

class Child extends Parent {
    // void showMessage() { // Compilation error: Cannot override final method
    //     System.out.println("Trying to override final method");
    // }
}

public class Main {
    public static void main(String[] args) {
        Child child = new Child();
        child.showMessage();
    }
}
```

Output:

```
This message cannot be overridden.
```



4. Final Classes in Java

A **final class** cannot be extended by any other class, ensuring that its behavior remains unchanged.

Example of a Final Class:

```
final class Utility {
    void displayMessage() {
        System.out.println("This class cannot be extended.");
    }
}

// class ExtendedUtility extends Utility { // Compilation error: Cannot inher
```



```
// }

public class Main {
    public static void main(String[] args) {
        Utility util = new Utility();
        util.displayMessage();
    }
}
```

Output:

This class cannot be extended.



5. Differences Between Final, Static, and Constant Variables

Feature	Final Variable	Static Variable	Constant Variable (<code>static final</code>)
Value Modification	Cannot be modified after initialization	Can be changed at any time	Cannot be modified after initialization
Memory Storage	Per instance (unless static)	Stored in class area	Stored in class area
Access	Requires an object (if non-static)	Accessed using class name	Accessed using class name
Example	<code>final int x = 10;</code>	<code>static int x = 1 0;</code>	<code>static final int X = 10;</code>

6. When to Use Final Members?

Use Case	Final Variable	Final Method	Final Class
Defining constants (e.g., <code>PI</code> , <code>MAX_USERS</code>)	<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No	<input type="checkbox"/> No



Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



Singleton Class in Java

Introduction

A **Singleton Class** in Java is a design pattern that restricts the instantiation of a class to a single instance. It ensures that only **one object** of the class exists in the entire application and provides a **global point of access** to that instance.

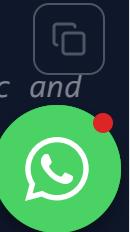
Why Use Singleton Pattern?

- To manage shared resources like database connections or configuration settings.
- To ensure controlled object creation and save memory.
- To provide global access to a single instance.

Implementing Singleton in Java (Basic Example)

Since we have already learned about **static** and **final**, we can use them to create a Singleton class effectively.

```
class Singleton {  
    private static final Singleton instance = new Singleton(); // Static and  
    private Singleton() {} // Private constructor to prevent instantiation
```



```
public static Singleton getInstance() {
    return instance; // Returns the single instance
}

public class Main {
    public static void main(String[] args) {
        Singleton obj1 = Singleton.getInstance();
        Singleton obj2 = Singleton.getInstance();
        System.out.println(obj1 == obj2); // Output: true (Same instance)
    }
}
```

Explanation:

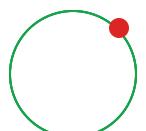
- **static** ensures that the instance is shared across the application.
- **final** ensures that the instance is assigned only once.
- **Private Constructor** prevents multiple object creation.

When to Use Singleton?

- ✓ Managing configuration settings.
- ✓ Handling logging frameworks.
- ✓ Database connection pooling.
- ✗ Not suitable for general object creation.

Conclusion

In this blog, we learned:



Java Tutorial – A Comprehensive Guide for Beginners

Introduction

Java is one of the most widely used and versatile programming languages in the world. From desktop applications to web services and mobile apps to enterprise systems, Java powers countless technologies and platforms.

This Java tutorial is designed for beginners who want to build a strong foundation in Java programming. Whether you're completely new to programming or transitioning from another language, this guide covers everything from core concepts to essential object-oriented principles and beyond.

If you're aiming to become a proficient Java developer, follow this structured path to master Java, step by step.

Prerequisites

Before you dive into Java, it's helpful (but not mandatory) to be familiar with:

- **Basic Computer Knowledge:** Understanding how software runs, what an IDE is, etc.
- **Logic Building:** Basic problem-solving and logical thinking.



- **Understanding of Programming Concepts** (optional): Knowing variables, loops, or functions from another language is a bonus.

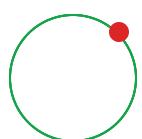
Master These Topics

Once you're ready, follow this structured list to learn Java from the ground up:

1. Introduction to Java
2. Java Fundamentals
3. Java Flow Control
4. Java Arrays
5. Java Methods
6. Java Object-Oriented Programming
7. Java Inner Classes
8. Static and Final Keywords
9. Java Exception Handling
10. Java Generics
11. Java Lambda Expressions
12. Java Collection Framework
13. Java Input/Output Streams
14. Java Reader/Writer

Introduction to Java

1. [Introduction to Java](#)
2. [Installing JDK on your computer](#)



[3. Java Comments](#)

Java Fundamentals

- [1. Java Variables and Literals](#)
- [2. Data Types in Java](#)
- [3. Operators in Java](#)
- [4. Java Basic Input and Output](#)
- [5. Java Expressions, Statements and Blocks](#)

Java Flow Control

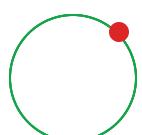
- [1. Java if else statement](#)
- [2. Java Ternary Operator](#)
- [3. Java For Loop](#)
- [4. Java while and do while loop](#)
- [5. Java continue and break statement](#)
- [6. Java Switch statement](#)

Java Arrays

- [1. Java Arrays](#)
- [2. Java Multidimensional Arrays](#)
- [3. Java Copy Arrays](#)

Java Methods

- [1. Methods in Java](#)



2. Parameter Passing in Java
3. Method Overloading
4. Variable Arguments
5. Java CommandLine Arguments
6. Java Recursive Method

Java Object-Oriented Programming

1. Principles of Object-Oriented Programming
2. Java Class and Objects
3. Java Constructor
4. Encapsulation and Data Hiding
5. Inheritance in Java
6. Constructors in Inheritance
7. this Vs super Keyword
8. Method Overriding
9. Dynamic Method Dispatch
10. Polymorphism using Overloading and Overriding
11. Abstract Classes
12. Interfaces in Java

Java Inner Classes

1. Inner Classes in Java





Java Exception Handling

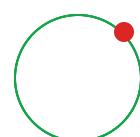
1. [What are Exceptions](#)
2. [How to handle Exception](#)
3. [Try and Catch Block](#)
4. [Multiple and Nested Try Catch](#)
5. [Class Exception](#)
6. [Checked and Unchecked Exceptions](#)
7. [Throw Vs Throws](#)
8. [Finally Block](#)
9. [Try with Resources](#)

Java Generics

1. [Introduction to Generics](#)
2. [Defining Generic Class](#)
3. [Bounds on Generics](#)
4. [Java Generic Methods](#)

Java Lambda Expressions

1. [Introduction to Lambda Expressions](#)
2. [Parameters in Lambda Expressions](#)



3. Capture in Lambda Expression

4. Method References

Java Collection Framework

1. Collections in Java

2. Understanding Collection Interface

3. Understanding List Interface in Java

4. Java ArrayList

5. Java LinkedList

6. Java Vector

7. Java Stack

8. Understanding Queue Interface in Java

9. Java PriorityQueue

10. Java ArrayDeque

11. Understanding Set Interface in Java

12. Java HashSet

13. Java LinkedHashSet

14. Java TreeSet

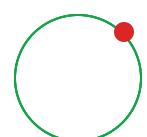
15. Understanding Map Interface in Java

16. Java HashMap

17. Java LinkedHashMap

18. Java TreeMap

19. Comparable Interface in Java



20. Comparator Interface in Java

Java Input/Output Streams

1. [What are Streams](#)
2. [InputStream and OutputStream Classes](#)
3. [FileInputStream and FileOutputStream](#)
4. [ByteArrayInputStream and ByteArrayOutputStream](#)
5. [ObjectInputStream and ObjectOutputStream](#)
6. [BufferedInputStream and BufferedOutputStream](#)
7. [Java PrintStream Class](#)

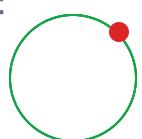
Java Reader/Writer

1. [Java File Class](#)
2. [Java Reader and Writer Class](#)
3. [InputStreamReader and OutputStreamWriter Class](#)
4. [FileReader and FileWriter Class](#)
5. [BufferedReader and BufferedWriter Class](#)
6. [StringReader and StringWriter Class](#)
7. [Java PrintWriter Class](#)

Key Features of Java

Java has remained popular for decades for good reason. Here's what makes it special:

- **Platform Independent:** Write once, run anywhere (thanks to the JVM).





 Pay Day Sale is Now Live!  Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



 Java Programming Handbook



Exceptions in Java

Introduction

When writing Java programs, errors can occur for various reasons, such as invalid user input, file handling issues, or network failures. These errors, when left unhandled, can cause the program to crash. This is where **exceptions** come into play. Java provides a robust mechanism to handle such unexpected situations, ensuring the application continues running smoothly.

In this blog, we will explore **what exceptions are**, why they occur, and how they help in writing error-free programs.

What is an Exception?

An **exception** in Java is an **unexpected event** that occurs during the execution of a program, disrupting its normal flow. Exceptions typically occur when a program encounters a situation it cannot handle, such as:

- Dividing a number by zero
- Accessing an invalid index in an array
- Opening a file that does not exist
- Attempting to convert a string to a number when it's not formatted correctly



Example of an Exception:

Let's consider an example where we try to divide a number by zero:

```
public class ExceptionExample {  
    public static void main(String[] args) {  
        int a = 10, b = 0;  
        int result = a / b; // This will cause an exception  
        System.out.println(result);  
    }  
}
```



Output:

```
Exception in thread "main" java.lang.ArithmetiException: / by zero
```



Since dividing by zero is not allowed, Java throws an **ArithmetiException** and terminates the program.

Why Do Exceptions Occur?

Exceptions in Java can occur due to various reasons, including:

- **Logical Errors:** Mistakes in the program logic (e.g., incorrect calculations).
- **Resource Issues:** Problems accessing files, databases, or memory.
- **Invalid User Input:** Trying to process incorrect input types.
- **Network Failures:** When a network connection is lost unexpectedly.

By handling exceptions properly, we can prevent these issues from **crashing the program** and ensure smooth execution.



How Java Handles Exceptions

To handle exceptions, Java provides a powerful mechanism using:

- **Try-Catch Blocks** – To catch and handle exceptions.
- **Finally Block** – To execute cleanup code regardless of exceptions.
- **Throw Keyword** – To manually trigger exceptions.
- **Throws Keyword** – To declare exceptions that a method might throw.
- **Try-With-Resources** – To manage resources like file handling safely.

We will explore each of these in detail in upcoming blogs.

Types of Errors in Java

Before diving deeper into exceptions, let's understand **two major categories of errors** in Java:

1. Compile-Time Errors

Compile-time errors occur **before the program runs**, preventing it from compiling successfully.

These include:

- **Syntax errors** (e.g., missing semicolons, misspelled keywords)
- **Type mismatches** (e.g., assigning a string to an integer variable)

Example:



```
public class CompileErrorExample {  
    public static void main(String[] args) {  
        int num = "hello"; // Error: Type mismatch  
    }  
}
```

2. Runtime Errors (Exceptions)

Runtime errors occur **during program execution**. These are called **exceptions** and can be handled using Java's exception-handling mechanism.

Example:

```
public class RuntimeErrorExample {  
    public static void main(String[] args) {  
        int[] numbers = {1, 2, 3};  
        System.out.println(numbers[5]); // Accessing an invalid index  
    }  
}
```

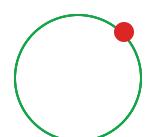
Output:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 5
```

Importance of Exception Handling

Without exception handling, a single error can cause the program to crash, leading to a poor user experience. Exception handling helps:

- **Improve program stability** by preventing unexpected crashes.
- **Provide meaningful error messages** to help in debugging.





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



How to Handle Exception in Java

Introduction

In our previous blog, we discussed what exceptions are and why they occur. Now, let's explore how to handle these exceptions in Java so that our programs don't crash unexpectedly.

Java provides several mechanisms to handle exceptions efficiently. In this blog, we will cover:

- Why exception handling is necessary
- Using **try-catch** blocks
- The **finally** block
- The **throw** and **throws** keywords
- Best practices for exception handling

Why Handle Exceptions?

Exception handling ensures that a program runs smoothly even when an error occurs. Without exception handling, the program will terminate abruptly when an exception arises. Handling exceptions properly allows us to:

- Prevent application crashes



- Display meaningful error messages to users
- Ensure proper resource management (closing files, database connections, etc.)
- Debug errors effectively

Using Try-Catch Blocks

The most common way to handle exceptions in Java is by using **try** and **catch** blocks. The **try block** contains the code that may cause an exception, and the **catch block** handles the exception if it occurs.

Syntax:

```
try {  
    // Code that may cause an exception  
} catch (ExceptionType e) {  
    // Handling the exception  
}
```

Example:

```
public class ExceptionHandlingExample {  
    public static void main(String[] args) {  
        try {  
            int result = 10 / 0; // This will cause ArithmeticException  
            System.out.println(result);  
        } catch (ArithmaticException e) {  
            System.out.println("Error: Cannot divide by zero.");  
        }  
    }  
}
```

Output:

Error: Cannot divide by zero.



In this example, instead of crashing, the program gracefully handles the division by zero error and displays a message.

Using Multiple Catch Blocks

Sometimes, different types of exceptions can occur. We can use multiple `catch` blocks to handle them separately.

Example:

```
public class MultipleCatchExample {  
    public static void main(String[] args) {  
        try {  
            int[] numbers = {1, 2, 3};  
            System.out.println(numbers[5]); // ArrayIndexOutOfBoundsException  
        } catch (ArithmaticException e) {  
            System.out.println("Arithmatic Exception occurred.");  
        } catch (ArrayIndexOutOfBoundsException e) {  
            System.out.println("Array Index Out of Bounds Exception occurred.  
        }  
    }  
}
```

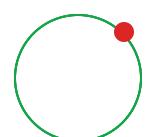


Output:

Array Index Out of Bounds Exception occurred.



The Finally Block



The **finally** block contains code that **always executes**, whether an exception occurs or not. It is commonly used for **cleanup operations**, such as closing files or releasing resources.

Example:

```
public class FinallyExample {  
    public static void main(String[] args) {  
        try {  
            int num = 10 / 0;  
        } catch (ArithmetricException e) {  
            System.out.println("Exception caught.");  
        } finally {  
            System.out.println("This will always execute.");  
        }  
    }  
}
```



Output:

```
Exception caught.  
This will always execute.
```

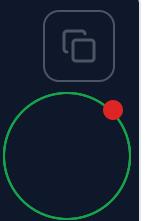


Using **throw** to Manually Throw Exceptions

Java allows us to manually throw exceptions using the **throw** keyword. This is useful when we want to create custom error messages.

Example:

```
public class ThrowExample {  
    public static void main(String[] args) {  
        int age = 15;  
        if (age < 18) {  
            throw new ArithmetricException("Age must be 18 or older");  
        }  
    }  
}
```



```
        throw new ArithmeticException("Age must be 18 or above.");
    }
    System.out.println("Access granted.");
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: Age must be 18 or a
```

Using throws in Method Declarations

The **throws** keyword is used in method declarations to indicate that a method might throw an exception. It **does not handle** the exception but **informs** the caller of the method that an exception may occur.

Example:

```
public class ThrowsExample {
    static void checkAge(int age) throws ArithmeticException {
        if (age < 18) {
            throw new ArithmeticException("Age must be 18 or above.");
        }
    }

    public static void main(String[] args) {
        checkAge(15);
    }
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: Age must be 18 or a
```



Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Try and Catch Block in Java

Introduction

In the previous blog, we introduced exception handling in Java and briefly discussed the **try-catch** mechanism. Now, let's take a deeper dive into how the **try** and **catch** blocks work, along with best practices and real-world examples.

In this blog, we will cover:

- What is a **try** block?
- What is a **catch** block?
- How **try-catch** works internally
- Handling multiple exceptions
- Best practices for using **try-catch**

What is a try Block?

A **try** block contains the **risky code** that might generate an exception. It allows Java to test a section of code for potential exceptions without stopping the execution of the program.

Syntax:



```
try {  
    // Code that may cause an exception  
}
```

Example:

```
public class TryExample {  
    public static void main(String[] args) {  
        try {  
            int number = 10 / 0; // Risky code (Division by zero)  
            System.out.println(number);  
        }  
    }  
}
```

Output:

```
Exception in thread "main" java.lang.ArithmetiException: / by zero
```

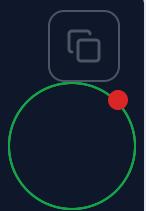
👉 Notice that the program crashes because there's no **catch** block to handle the exception.

What is a **catch** Block?

A **catch** block is used to handle exceptions thrown from the **try** block. It prevents the program from crashing by executing alternative code when an exception occurs.

Syntax:

```
try {  
    // Code that may cause an exception  
} catch (ExceptionType e) {  
    // Code to handle the exception
```



Example:

```
public class TryCatchExample {  
    public static void main(String[] args) {  
        try {  
            int number = 10 / 0;  
        } catch (ArithmaticException e) {  
            System.out.println("Cannot divide by zero!");  
        }  
        System.out.println("Program continues...");  
    }  
}
```



Output:

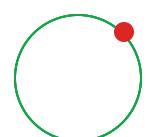
```
Cannot divide by zero!  
Program continues...
```



👉 Now, the exception is handled, and the program doesn't crash.

How try-catch Works Internally

1. The program starts executing the **try** block.
2. If no exception occurs, the **catch** block is **skipped**.
3. If an exception occurs:
 - Execution immediately jumps to the **catch** block.
 - The rest of the **try** block is **skipped**.
4. After handling the exception, the program continues executing normally.



Here's an example where no exception occurs:

```
public class NoExceptionExample {  
    public static void main(String[] args) {  
        try {  
            int num = 10 / 2;  
            System.out.println("Result: " + num);  
        } catch (ArithmaticException e) {  
            System.out.println("An error occurred.");  
        }  
        System.out.println("Program finished.");  
    }  
}
```

Output:

```
Result: 5  
Program finished.
```

👉 The **catch** block is skipped because no error occurred.

Handling Multiple Exceptions

Sometimes, multiple exceptions may occur in a single **try** block. Java allows handling them using multiple **catch** blocks.

Example:

```
public class MultipleCatchExample {  
    public static void main(String[] args) {  
        try {  
            int[] numbers = {1, 2, 3};  
            System.out.println(numbers[5]); // ArrayIndexOutOfBoundsException  
        } catch (ArithmaticException e) {  
        }  
    }  
}
```

```
        System.out.println("Arithmetic error occurred.");
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Array index is out of bounds.");
    }
}
```

Output:

```
Array index is out of bounds.
```



👉 Java checks each **catch** block sequentially to find a match for the exception type.

Catching Multiple Exceptions in One catch Block (Java 7+)

Instead of writing multiple **catch** blocks, Java allows handling multiple exceptions in a **single catch** block using the **|** (pipe) symbol.

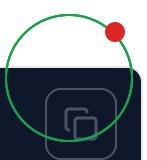
Example:

```
public class MultiCatchExample {
    public static void main(String[] args) {
        try {
            int num = Integer.parseInt("abc"); // NumberFormatException
        } catch (ArithmaticException | NumberFormatException e) {
            System.out.println("An error occurred: " + e.getMessage());
        }
    }
}
```



Output:

```
An error occurred: For input string: "abc"
```





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



Multiple and Nested Try-Catch Blocks in Java

Introduction

In the previous blog, we covered the basics of **try** and **catch** blocks. Now, let's explore two advanced concepts:

1. **Multiple Try-Catch Blocks** – Handling different exceptions separately.
2. **Nested Try-Catch Blocks** – Handling exceptions within another **try** block.

By the end of this blog, you'll understand how to handle different exceptions efficiently and structure your error-handling logic effectively.

Multiple Try-Catch Blocks

A **try** block can throw different types of exceptions. Java allows multiple **catch** blocks to handle different exceptions separately.

Syntax:

```
try {  
    // Risky code  
} catch (ExceptionType1 e1) {  
    // Handle ExceptionType1
```



```
} catch (ExceptionType2 e2) {  
    // Handle ExceptionType2  
} catch (Exception e) {  
    // Handle any other exceptions
```

Example:

```
public class MultipleTryCatchExample {  
    public static void main(String[] args) {  
        try {  
            int[] numbers = {1, 2, 3};  
            int result = 10 / 0; // ArithmeticException  
            System.out.println(numbers[5]); // ArrayIndexOutOfBoundsException  
        } catch (ArithmaticException e) {  
            System.out.println("Error: Cannot divide by zero.");  
        } catch (ArrayIndexOutOfBoundsException e) {  
            System.out.println("Error: Array index is out of bounds.");  
        } catch (Exception e) {  
            System.out.println("A general exception occurred: " + e.getMessage())  
        }  
    }  
}
```

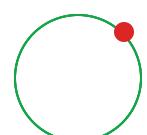
Output:

```
Error: Cannot divide by zero.
```

👉 Notice that the second exception (`ArrayIndexOutOfBoundsException`) never occurs because the first error stops execution.

Best Practice: Always catch specific exceptions first, then use a generic `Exception` catch block at the end.

Nested Try-Catch Blocks



A **try** block inside another **try** block is called a **nested try-catch**. This is useful when handling exceptions at different levels in a program.

Why Use Nested Try-Catch?

- When you have operations that can fail independently.
- To handle exceptions at different scopes in a program.
- To prevent a single error from stopping the entire program.

Syntax:

```
try {  
    try {  
        // Inner risky code  
    } catch (ExceptionType1 e) {  
        // Handle specific exception  
    }  
} catch (ExceptionType2 e) {  
    // Handle outer exception  
}
```

Example:

```
public class NestedTryCatchExample {  
    public static void main(String[] args) {  
        try {  
            System.out.println("Outer try block");  
            try {  
                int[] numbers = {1, 2, 3};  
                System.out.println(numbers[5]); // ArrayIndexOutOfBoundsException  
            } catch (ArrayIndexOutOfBoundsException e) {  
                System.out.println("Inner catch: Array index out of bounds.")  
            }  
            int result = 10 / 0; // ArithmeticException (outer block)  
        } catch (ArithmeticException e) {  
            System.out.println("Outer catch: Arithmetic exception")  
        }  
    }  
}
```

```
        System.out.println("Outer catch: Cannot divide by zero.");
    }
    System.out.println("Program continues...");
}
}
```

Output:

```
Outer try block
Inner catch: Array index out of bounds.
Outer catch: Cannot divide by zero.
Program continues...
```



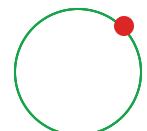
👉 The inner exception ([ArrayIndexOutOfBoundsException](#)) is handled separately from the outer one ([ArithmetcException](#)).

Key Differences Between Multiple and Nested Try-Catch

Feature	Multiple Try-Catch	Nested Try-Catch
Structure	Multiple <code>catch</code> blocks after a single <code>try</code> block	One <code>try-catch</code> inside another <code>try</code> block
Handling Scope	Handles different exceptions independently	Handles exceptions at different levels
Use Case	When multiple errors may occur in a single block	When errors occur at different levels of execution

Best Practices for Nested and Multiple Try-Catch

- Use multiple catch blocks when different exceptions need separate handling.





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Class Exception in Java

Introduction

In Java, exceptions are objects that represent an error or an unexpected behavior in a program. These exceptions belong to the class hierarchy that extends `java.lang.Throwable`. In this blog, we will explore:

- The Exception class hierarchy
- Commonly used methods in the Exception class
- Creating custom exceptions in Java

By the end of this blog, you'll have a strong understanding of how Java organizes exceptions and how you can extend the `Exception` class to create your own exceptions.

Exception Class Hierarchy in Java

Java organizes exceptions into a well-structured hierarchy under the `Throwable` class:

```
Throwable
|
+-- Exception
|   +-- IOException
|   +-- SQLException
```



```
|   └─ ClassNotFoundException  
|   └─ RuntimeException  
|       └─ ArithmeticException  
|       └─ NullPointerException  
|       └─ ArrayIndexOutOfBoundsException  
  
└─ Error  
    └─ StackOverflowError  
    └─ OutOfMemoryError
```

- **Throwable**: The root class of all exceptions and errors.
- **Exception**: Represents errors that can be recovered from.
- **RuntimeException**: Represents unchecked exceptions.
- **Error**: Represents serious issues that usually cannot be handled.

Commonly Used Methods in the Exception Class

The **Exception** class provides several useful methods to get information about an exception:

Method	Description
getMessage()	Returns a detailed message of the exception.
toString()	Returns a string representation of the exception.
printStackTrace()	Prints the complete stack trace of the exception.

Example:

```
public class ExceptionMethodsExample {  
    public static void main(String[] args) {  
        try {  
            int result = 10 / 0; // This will throw ArithmeticException
```



```
        } catch (ArithmException e) {
            System.out.println("Message: " + e.getMessage());
            System.out.println("toString(): " + e.toString());
            System.out.print("StackTrace: ");
            e.printStackTrace();
        }
    }
}
```

Output:

```
Message: / by zero
toString(): java.lang.ArithmException: / by zero
StackTrace: java.lang.ArithmException: / by zero
at ExceptionMethodsExample.main(ExceptionMethodsExample.java:5)
```

Creating Custom Exceptions in Java

Java allows developers to create their own exception classes by extending **Exception**.

When Should You Create Custom Exceptions?

- When built-in exceptions do not represent a specific error scenario in your application.
- When you want to define meaningful exception names for clarity.
- When you need additional fields/methods in your exception class.

Example: Custom Exception

```
class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}
```

```
public class CustomExceptionExample {  
    public static void validateAge(int age) throws InvalidAgeException {  
        if (age < 18) {  
            throw new InvalidAgeException("Age must be 18 or above to vote.");  
        }  
        System.out.println("Valid age for voting.");  
    }  
  
    public static void main(String[] args) {  
        try {  
            validateAge(16);  
        } catch (InvalidAgeException e) {  
            System.out.println("Caught Exception: " + e.getMessage());  
        }  
    }  
}
```

Output:

Caught Exception: Age must be 18 or above to vote.

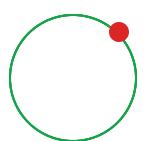


Best Practices for Using Exception Classes

- Use meaningful exception names – `InvalidAgeException` is clearer than `Exception`.
- Extend `Exception` for checked exceptions and `RuntimeException` for unchecked exceptions.
- Always provide descriptive error messages in custom exceptions.
- Use built-in exceptions whenever possible instead of creating new ones unnecessarily.

Conclusion

In this blog, we learned:





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



Checked and Unchecked Exceptions in Java

Introduction

In Java, exceptions are categorized into two types:

1. **Checked Exceptions** – Exceptions that must be handled at compile time.
2. **Unchecked Exceptions** – Exceptions that occur at runtime and do not require mandatory handling.

In this blog, we will discuss:

- The difference between checked and unchecked exceptions
- Common examples of both
- How to handle them properly
- Best practices for using exceptions in Java

What are Checked Exceptions?

Checked exceptions are exceptions that **must be handled** using a **try-catch** block or declared using **throws** in the method signature. If not handled, the compiler will throw an error.



Examples of Checked Exceptions:

- **IOException**
- **SQLException**
- **FileNotFoundException**
- **ClassNotFoundException**
- **InterruptedException**

Example of Checked Exception:

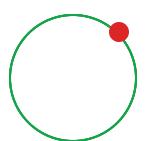
```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class CheckedExceptionExample {
    public static void main(String[] args) {
        try {
            File file = new File("test.txt");
            FileReader fr = new FileReader(file); // This may throw FileNotFoundException
        } catch (IOException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }
    }
}
```

Output:

```
Exception caught: test.txt (No such file or directory)
```

Explanation: Since reading a file is an external operation, Java forces us to handle **IOException** to avoid crashes.



What are Unchecked Exceptions?

Unchecked exceptions are **runtime exceptions** that occur due to programming errors. They are **not checked at compile time**, meaning the developer is responsible for handling them.

Examples of Unchecked Exceptions:

- **NullPointerException**
- **ArrayIndexOutOfBoundsException**
- **ArithmeticeException**
- **NumberFormatException**
- **IllegalArgumentException**

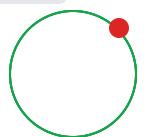
Example of Unchecked Exception:

```
public class UncheckedExceptionExample {  
    public static void main(String[] args) {  
        int a = 10, b = 0;  
        int result = a / b; // This will throw ArithmeticeException  
        System.out.println("Result: " + result);  
    }  
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticeException: / by zero  
at UncheckedExceptionExample.main(UncheckedExceptionExample.java:4)
```

Explanation: Dividing by zero is an invalid operation, leading to an **ArithmeticeException** at runtime.



Key Differences Between Checked and Unchecked Exceptions

Feature	Checked Exception	Unchecked Exception
Checked at Compile Time	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> No
Must be Handled	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> No
Extends	<code>Exception</code> (but NOT <code>RuntimeException</code>)	<code>RuntimeException</code>
Example	<code>IOException</code> , <code>SQLException</code>	<code>NullPointerException</code> , <code>ArithmaticException</code>
Occurs Due to	External factors (e.g., file missing, database connection error)	Programming mistakes (e.g., null references, division by zero)

How to Handle Checked and Unchecked Exceptions

Handling Checked Exceptions:

Checked exceptions must be handled using `try-catch` or declared with `throws`.

```
import java.io.IOException;

public class CheckedHandlingExample {
    public static void main(String[] args) {
        try {
            throw new IOException("File not found");
        } catch (IOException e) {
            System.out.println("Handled checked exception: " + e.getMessage());
        }
    }
}
```

Handling Unchecked Exceptions:

Unchecked exceptions **should be prevented** through proper coding practices. However, you can use **try-catch** if needed.

```
public class UncheckedHandlingExample {  
    public static void main(String[] args) {  
        try {  
            String str = null;  
            System.out.println(str.length()); // Will throw NullPointerException  
        } catch (NullPointerException e) {  
            System.out.println("Handled unchecked exception: " + e.getMessage)  
        }  
    }  
}
```

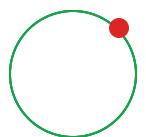


Best Practices for Using Exceptions

- Use checked exceptions for recoverable conditions (e.g., missing files, invalid input).
- Use unchecked exceptions for programming logic errors (e.g., null values, division by zero).
- Avoid unnecessary exception handling – only catch exceptions when you can recover from them.
- Use meaningful exception messages to help debug issues quickly.
- Always close resources properly (use **try-with-resources** for handling resources like files, sockets, etc.).

Conclusion

In this blog, we covered:





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



Throw vs Throws in Java

Introduction

When handling exceptions in Java, developers often come across the **throw** and **throws** keywords. Though they look similar, their usage and purpose are entirely different. In this blog, we will explore:

- The difference between **throw** and **throws**
- When to use each
- Examples illustrating their usage
- Best practices

Understanding throw

The **throw** keyword is used **inside a method** to explicitly throw an exception. It allows us to create and throw both checked and unchecked exceptions manually.

Syntax:

```
throw new ExceptionType("Error message");
```



Example of throw:

```
public class ThrowExample {  
    public static void validateAge(int age) {  
        if (age < 18) {  
            throw new IllegalArgumentException("Age must be 18 or above");  
        }  
        System.out.println("Valid age");  
    }  
  
    public static void main(String[] args) {  
        validateAge(15); // This will throw an exception  
    }  
}
```

Output:

```
Exception in thread "main" java.lang.IllegalArgumentException: Age must be 18  
at ThrowExample.validateAge(ThrowExample.java:4)  
at ThrowExample.main(ThrowExample.java:9)
```

Explanation: The **throw** keyword is used inside **validateAge()** to manually throw an **IllegalArgumentException** if the age is below 18.

Understanding throws

The **throws** keyword is used in a method declaration to specify that the method might throw an exception. This allows the caller of the method to handle the exception.

Syntax:

```
returnType methodName(parameters) throws ExceptionType {  
    // Method body  
}
```

Example of throws:

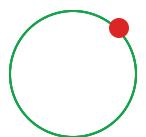
```
import java.io.IOException;  
  
public class ThrowsExample {  
    public static void readFile() throws IOException {  
        throw new IOException("File not found");  
    }  
  
    public static void main(String[] args) {  
        try {  
            readFile();  
        } catch (IOException e) {  
            System.out.println("Exception caught: " + e.getMessage());  
        }  
    }  
}
```

Output:

```
Exception caught: File not found
```

Explanation: Here, `readFile()` declares that it throws an `IOException`. The calling method (`main`) must handle it using a `try-catch` block.

Key Differences Between throw and throws



Feature	<code>throw</code>	<code>throws</code>
Purpose	Used to explicitly throw an exception	Declares that a method might throw exceptions
Placement	Inside a method body	In the method signature
Type	Used for both checked and unchecked exceptions	Mainly used for checked exceptions
Handling	Exception must be caught or declared by the caller	Caller must handle the exception

Example: `throw` vs `throws`

```
import java.io.IOException;

public class ThrowVsThrowsExample {
    // Using throws in method declaration
    public static void riskyMethod() throws IOException {
        throw new IOException("Risky operation failed");
    }

    public static void main(String[] args) {
        try {
            riskyMethod();
        } catch (IOException e) {
            System.out.println("Exception handled: " + e.getMessage());
        }
    }
}
```

Output:

Exception handled: Risky operation failed



Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Finally Block in Java

Introduction

Exception handling in Java provides several constructs like **try**, **catch**, **throw**, and **throws**. However, in many cases, we need to ensure that some code executes regardless of whether an exception occurs or not. This is where the **finally** block comes into play.

In this blog, we will cover:

- What is the **finally** block?
- Why is it needed?
- How does it work with **try-catch**?
- Practical examples
- Best practices

What is the **finally** Block?

The **finally** block in Java is used to execute important code after the try-catch block. It runs regardless of whether an exception is thrown or caught, making it useful for resource cleanup like closing database connections, file streams, or network sockets.



Key Features:

- The `finally` block is **optional** but highly recommended.
- It always executes, **even if an exception is not thrown**.
- It runs **even if there is a `return` statement** inside the `try` or `catch` block.

Syntax of `finally` Block

```
try {  
    // Code that may throw an exception  
} catch (ExceptionType e) {  
    // Handle exception  
} finally {  
    // Cleanup code that always executes  
}
```

Example: Using `finally` with Exception Handling

```
public class FinallyExample {  
    public static void main(String[] args) {  
        try {  
            int result = 10 / 0; // This will cause an exception  
        } catch (ArithmaticException e) {  
            System.out.println("Exception caught: " + e.getMessage());  
        } finally {  
            System.out.println("Finally block executed");  
        }  
    }  
}
```

Output:

```
Exception caught: / by zero  
Finally block executed
```



Explanation: The **finally** block executes after handling the **ArithmaticException**, ensuring that the necessary cleanup operations are performed.

Example: finally Without an Exception

```
public class FinallyNoException {  
    public static void main(String[] args) {  
        try {  
            System.out.println("Try block executed");  
        } catch (Exception e) {  
            System.out.println("Exception caught");  
        } finally {  
            System.out.println("Finally block executed");  
        }  
    }  
}
```



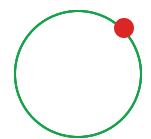
Output:

```
Try block executed  
Finally block executed
```



Explanation: Since no exception is thrown, the **catch** block is skipped, but the **finally** block still executes.

Example: finally with return Statement



Even if a `return` statement is present in the `try` block, the `finally` block still executes before the method exits.

```
public class FinallyWithReturn {  
    public static int getNumber() {  
        try {  
            return 10;  
        } finally {  
            System.out.println("Finally block executed before returning");  
        }  
    }  
  
    public static void main(String[] args) {  
        System.out.println("Returned value: " + getNumber());  
    }  
}
```

Output:

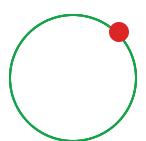
```
Finally block executed before returning  
Returned value: 10
```

Explanation: Even though `return 10;` is inside the `try` block, the `finally` block executes before the return statement takes effect.

When Does the `finally` Block NOT Execute?

The `finally` block will not execute in the following cases:

1. If the JVM terminates the program using `System.exit(0);`
2. If a fatal error occurs (like StackOverflowError or OutOfMemoryError)



Example:

```
public class FinallyNotExecuted {  
    public static void main(String[] args) {  
        try {  
            System.out.println("Inside try block");  
            System.exit(0); // Terminates JVM  
        } finally {  
            System.out.println("Finally block (won't execute)");  
        }  
    }  
}
```

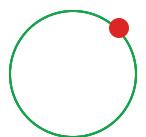
Output:

```
Inside try block
```

Explanation: Since `System.exit(0);` terminates the JVM, the `finally` block does not get a chance to execute.

Best Practices for Using `finally`

- Always use `finally` when working with external resources (files, databases, network connections) to ensure proper cleanup.
- Avoid placing **return statements** inside `finally`, as it can override the return value of the method.
- Be cautious while using `System.exit(0);` as it prevents the `finally` block from executing.
- Don't include complex logic in `finally`; keep it simple for cleanup purposes.





 Pay Day Sale is Now Live!  Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



 Java Programming Handbook



Try-with-Resources in Java

Introduction

In Java, managing resources like files, database connections, and sockets requires proper handling to avoid memory leaks. Traditionally, resources were closed using `finally` blocks, but this approach can be error-prone and verbose. Java introduced Try-with-Resources to simplify resource management.

In this blog, we will cover:

- What is Try-with-Resources?
- How it works
- Advantages over traditional exception handling
- Examples and best practices

What is Try-with-Resources?

Try-with-Resources (introduced in Java 7) is a feature that allows automatic closing of resources when they are no longer needed. It ensures that resources are closed at the end of the `try` block, reducing the risk of resource leaks.



Key Features:

- Works with any resource that implements `AutoCloseable` (such as `FileReader`, `BufferedReader`, `Connection`, etc.).
- Automatically closes resources without needing an explicit `finally` block.
- Reduces boilerplate code and improves readability.

Syntax of Try-with-Resources

```
try (ResourceType resource = new ResourceType()) {  
    // Use the resource  
} catch (ExceptionType e) {  
    // Handle exception  
}
```

Key Points:

- The resource is **declared inside parentheses** after the `try` keyword.
- It is **automatically closed** when the `try` block exits.
- No need for an explicit `finally` block to close the resource.

Example: Try-with-Resources with File Handling

```
import java.io.BufferedReader;  
import java.io.FileReader;  
import java.io.IOException;  
  
public class TryWithResourcesExample {  
    public static void main(String[] args) {  
        try (BufferedReader reader = new BufferedReader(new FileReader("example.txt"))){  
            System.out.println("File content: " + reader.readLine());  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
        } catch (IOException e) {
            System.out.println("Error reading file: " + e.getMessage());
        }
    }
```

Output (Assuming example.txt contains "Hello, World!"):

```
File content: Hello, World!
```



Explanation:

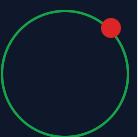
- The **BufferedReader** is declared inside **try**.
- It is automatically closed after execution, even if an exception occurs.

Example: Try-with-Resources with Database Connection

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class DatabaseExample {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/mydb";
        String user = "root";
        String password = "password";

        try (Connection conn = DriverManager.getConnection(url, user, password)) {
            Statement stmt = conn.createStatement();
            stmt.executeUpdate("INSERT INTO users (name, email) VALUES ('John Doe', 'john@example.com')");
            System.out.println("Data inserted successfully");
        } catch (SQLException e) {
            System.out.println("Database error: " + e.getMessage());
        }
    }
}
```



```
}
```

Key Benefits:

- **Connection** and **Statement** are automatically closed after execution.
- Prevents connection leaks and improves resource management.

Why Use Try-with-Resources Instead of Finally?

Feature	Traditional Finally Block	Try-with-Resources
Closes resources	Manually in <code>finally</code>	Automatically
Code readability	More verbose	Cleaner, less code
Risk of leaks	Higher if forget to close	Lower due to auto-close
Exception Handling	Requires extra try-catch	Handled within try

Example: Traditional Finally vs. Try-with-Resources

Using Finally Block:

```
BufferedReader reader = null;
try {
    reader = new BufferedReader(new FileReader("example.txt"));
    System.out.println(reader.readLine());
} catch (IOException e) {
    System.out.println("Error: " + e.getMessage());
} finally {
    try {
        if (reader != null) reader.close();
    } catch (IOException e) {
        System.out.println("Error closing file");
    }
}
```



Using Try-with-Resources:

```
try (BufferedReader reader = new BufferedReader(new FileReader("example.txt")) {
    System.out.println(reader.readLine());
} catch (IOException e) {
    System.out.println("Error: " + e.getMessage());
}
```

Conclusion: The Try-with-Resources approach is cleaner, avoids manual resource management, and reduces the risk of forgetting to close resources.

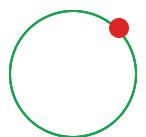
Best Practices for Try-with-Resources

- Use for handling files, database connections, network sockets, etc.
- Declare multiple resources in a **single try block** if needed.
- Prefer Try-with-Resources over manual **finally** block management.
- Ensure that the resource implements **AutoCloseable** or **Closeable**.

Conclusion

In this blog, we learned:

- The importance of Try-with-Resources in Java.
- How it simplifies resource management and avoids memory leaks.
- Examples of using it with file handling and databases.
- Why it is preferable over traditional **finally** blocks.



Java Tutorial – A Comprehensive Guide for Beginners

Introduction

Java is one of the most widely used and versatile programming languages in the world. From desktop applications to web services and mobile apps to enterprise systems, Java powers countless technologies and platforms.

This Java tutorial is designed for beginners who want to build a strong foundation in Java programming. Whether you're completely new to programming or transitioning from another language, this guide covers everything from core concepts to essential object-oriented principles and beyond.

If you're aiming to become a proficient Java developer, follow this structured path to master Java, step by step.

Prerequisites

Before you dive into Java, it's helpful (but not mandatory) to be familiar with:

- **Basic Computer Knowledge:** Understanding how software runs, what an IDE is, etc.
- **Logic Building:** Basic problem-solving and logical thinking.



- **Understanding of Programming Concepts** (optional): Knowing variables, loops, or functions from another language is a bonus.

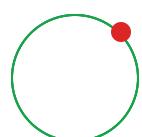
Master These Topics

Once you're ready, follow this structured list to learn Java from the ground up:

1. Introduction to Java
2. Java Fundamentals
3. Java Flow Control
4. Java Arrays
5. Java Methods
6. Java Object-Oriented Programming
7. Java Inner Classes
8. Static and Final Keywords
9. Java Exception Handling
10. Java Generics
11. Java Lambda Expressions
12. Java Collection Framework
13. Java Input/Output Streams
14. Java Reader/Writer

Introduction to Java

1. [Introduction to Java](#)
2. [Installing JDK on your computer](#)



[3. Java Comments](#)

Java Fundamentals

- [1. Java Variables and Literals](#)
- [2. Data Types in Java](#)
- [3. Operators in Java](#)
- [4. Java Basic Input and Output](#)
- [5. Java Expressions, Statements and Blocks](#)

Java Flow Control

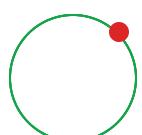
- [1. Java if else statement](#)
- [2. Java Ternary Operator](#)
- [3. Java For Loop](#)
- [4. Java while and do while loop](#)
- [5. Java continue and break statement](#)
- [6. Java Switch statement](#)

Java Arrays

- [1. Java Arrays](#)
- [2. Java Multidimensional Arrays](#)
- [3. Java Copy Arrays](#)

Java Methods

- [1. Methods in Java](#)



2. Parameter Passing in Java
3. Method Overloading
4. Variable Arguments
5. Java CommandLine Arguments
6. Java Recursive Method

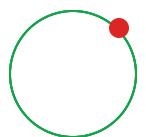
Java Object-Oriented Programming

1. Principles of Object-Oriented Programming
2. Java Class and Objects
3. Java Constructor
4. Encapsulation and Data Hiding
5. Inheritance in Java
6. Constructors in Inheritance
7. this Vs super Keyword
8. Method Overriding
9. Dynamic Method Dispatch
10. Polymorphism using Overloading and Overriding
11. Abstract Classes
12. Interfaces in Java

Java Inner Classes

1. Inner Classes in Java

Static and Final Keywords



1. Static Members and Static Blocks in Java
2. Final Members
3. Singleton Class

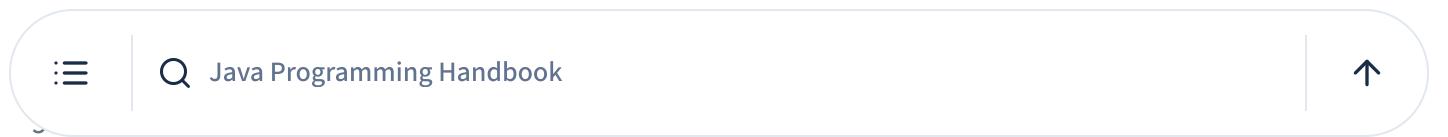
Java Exception Handling

1. What are Exceptions
2. How to handle Exception
3. Try and Catch Block
4. Multiple and Nested Try Catch

≡ Class Exception



Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)

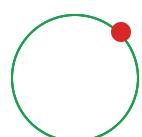


Java Generics

1. Introduction to Generics
2. Defining Generic Class
3. Bounds on Generics
4. Java Generic Methods

Java Lambda Expressions

1. Introduction to Lambda Expressions
2. Parameters in Lambda Expressions



3. Capture in Lambda Expression

4. Method References

Java Collection Framework

1. Collections in Java

2. Understanding Collection Interface

3. Understanding List Interface in Java

4. Java ArrayList

5. Java LinkedList

6. Java Vector

7. Java Stack

8. Understanding Queue Interface in Java

9. Java PriorityQueue

10. Java ArrayDeque

11. Understanding Set Interface in Java

12. Java HashSet

13. Java LinkedHashSet

14. Java TreeSet

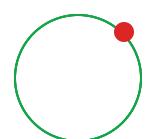
15. Understanding Map Interface in Java

16. Java HashMap

17. Java LinkedHashMap

18. Java TreeMap

19. Comparable Interface in Java



20. Comparator Interface in Java

Java Input/Output Streams

1. [What are Streams](#)
2. [InputStream and OutputStream Classes](#)
3. [FileInputStream and FileOutputStream](#)
4. [ByteArrayInputStream and ByteArrayOutputStream](#)
5. [ObjectInputStream and ObjectOutputStream](#)
6. [BufferedInputStream and BufferedOutputStream](#)
7. [Java PrintStream Class](#)

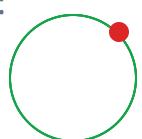
Java Reader/Writer

1. [Java File Class](#)
2. [Java Reader and Writer Class](#)
3. [InputStreamReader and OutputStreamWriter Class](#)
4. [FileReader and FileWriter Class](#)
5. [BufferedReader and BufferedWriter Class](#)
6. [StringReader and StringWriter Class](#)
7. [Java PrintWriter Class](#)

Key Features of Java

Java has remained popular for decades for good reason. Here's what makes it special:

- **Platform Independent:** Write once, run anywhere (thanks to the JVM).





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Introduction to Generics in Java

Introduction

Generics in Java allow us to create classes, interfaces, and methods with type parameters. This provides **type safety, code reusability, and eliminates type casting**. Before Generics, Java developers used objects of type **Object**, leading to runtime errors. Generics solve this problem by enabling compile-time type checking.

In this blog, we will cover:

- What are Generics?
- Why Use Generics?
- Advantages of Generics
- Basic Syntax of Generics
- Examples

What are Generics?

Generics allow you to create **type-safe** classes and methods by defining a placeholder for types. Instead of specifying a fixed data type, you can use a **type parameter** that is replaced by an actual type during runtime.



Example Without Generics (Before Java 5)

```
class DataContainer {  
    private Object data;  
  
    public void setData(Object data) {  
        this.data = data;  
    }  
  
    public Object getData() {  
        return data;  
    }  
}  
  
public class WithoutGenerics {  
    public static void main(String[] args) {  
        DataContainer container = new DataContainer();  
        container.setData("Hello");  
        String str = (String) container.getData(); // Requires explicit cast  
        System.out.println(str);  
    }  
}
```

Example With Generics

```
class GenericContainer<T> {  
    private T data;  
  
    public void setData(T data) {  
        this.data = data;  
    }  
  
    public T getData() {  
        return data;  
    }  
}
```

```

public class WithGenerics {
    public static void main(String[] args) {
        GenericContainer<String> container = new GenericContainer<>(); // Type safety
        container.setData("Hello Generics");

        String str = container.getData(); // No casting required
        System.out.println(str);
    }
}

```

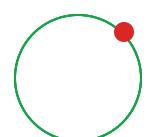
- With Generics:** No type casting, better readability, and type safety.

Why Use Generics?

- Type Safety** – Generics ensure that only the specified type is allowed in an instance.
- Code Reusability** – The same class or method can be used with different data types.
- No Need for Type Casting** – Avoids unnecessary explicit casting of objects.
- Compile-Time Checking** – Detects type-related errors at compile-time rather than runtime.

Advantages of Generics

Feature	Without Generics	With Generics
Type Safety	✗ No	✓ Yes
Type Casting	✗ Required	✓ Not Required
Code Reusability	✗ No	✓ Yes
Performance	✗ Slower (due to casting)	✓ Faster



Feature	Without Generics	With Generics
Compile-Time Checking	No	Yes

Basic Syntax of Generics

Generic Class Syntax

```
class Box<T> { // T is a type parameter
    private T value;

    public void setValue(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }
}
```



Creating an Instance of a Generic Class

```
public class GenericExample {
    public static void main(String[] args) {
        Box<Integer> intBox = new Box<>();
        intBox.setValue(100);
        System.out.println(intBox.getValue());

        Box<String> strBox = new Box<>();
        strBox.setValue("Hello Generics");
        System.out.println(strBox.getValue());
    }
}
```



Key Point: The same **Box** class works for both **Integer** and **String** without modification.





 Pay Day Sale is Now Live!  Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



Defining Generic Classes in Java

Introduction

In the previous blog, we introduced **Generics in Java** and discussed how they improve type safety and code reusability. Now, we will dive deeper into **defining generic classes**, understanding how they work, and implementing them with examples.

In this blog, we will cover:

- What is a Generic Class?
- Syntax of Generic Classes
- Creating and Using Generic Classes
- Multiple Type Parameters
- Examples

What is a Generic Class?

A **Generic Class** is a class that is parameterized with a type. Instead of specifying a fixed type, we use a **type parameter** (e.g., **T**, **E**, **K**, **V**) which gets replaced with an actual type at runtime.

Syntax of Generic Classes



```
class ClassName<T> {  
    private T value; // Declare a variable of type T  
  
    public void setValue(T value) {  
        this.value = value;  
    }  
  
    public T getValue() {  
        return value;  
    }  
}
```



- ◆ **T** is a **type parameter** that will be replaced with an actual type when an object is created.

Creating and Using Generic Classes

Example 1: Single Type Parameter

```
class Box<T> {  
    private T content;  
  
    public void setContent(T content) {  
        this.content = content;  
    }  
  
    public T getContent() {  
        return content;  
    }  
}  
  
public class GenericClassExample {  
    public static void main(String[] args) {  
        Box<String> stringBox = new Box<>();  
        stringBox.setContent("Hello Generics");  
        System.out.println("String content: " + stringBox.getContent());  
  
        Box<Integer> intBox = new Box<>();
```



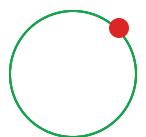
```
    intBox.setContent(100);
    System.out.println("Integer content: " + intBox.getContent());
}
}
```

- No explicit type casting needed!

Multiple Type Parameters

A generic class can have more than one type parameter.

Example 2: Generic Class with Two Type Parameters





```
class Pair<K, V> {
    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() {
        return key;
    }

    public V getValue() {
        return value;
    }
}

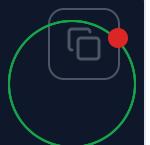
public class MultipleGenericsExample {
    public static void main(String[] args) {
        Pair<String, Integer> pair = new Pair<>("Age", 25);
        System.out.println("Key: " + pair.getKey() + ", Value: " + pair.getValue());
    }
}
```

Supports multiple types in a single class!

Wildcards in Generics

Wildcards allow a generic class to accept multiple unknown types.

- ◆ Example: Accepting any type using `<?>`



```
class Printer<T> {
    private T data;
```

```

public class Printer<T data> {
    this.data = data;
}

public void printData() {
    System.out.println("Data: " + data);
}
}

public class WildcardExample {
    public static void main(String[] args) {
        Printer<Integer> intPrinter = new Printer<>(123);
        Printer<String> strPrinter = new Printer<>("Generics Example");

        printAnyType(intPrinter);
        printAnyType(strPrinter);
    }

    public static void printAnyType(Printer<?> printer) {
        printer.printData();
    }
}

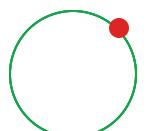
```

Allows flexibility while maintaining type safety.

Conclusion

In this blog, we learned:

- How to define Generic Classes
- The syntax for creating Generic Classes
- Using multiple type parameters in a class
- The use of wildcards (`<?>`) to handle unknown types





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



Bounds on Generics in Java

Introduction

In the previous blog, we learned how to define **Generic Classes** in Java. While generics allow flexibility, sometimes we need to **restrict** the type parameters to specific types. This is where **bounded type parameters** come into play.

In this blog, we will cover:

- What are Bounded Type Parameters?
- Upper Bounds (`extends` keyword)
- Lower Bounds (`super` keyword)
- Why and When to Use Bounds?
- Examples

What are Bounded Type Parameters?

By default, a generic type parameter (`T`) can be **any type**. But in some cases, we may want to limit `T` to **only certain types** (e.g., only `Number` or its subclasses). This is called **bounding a type parameter**.



◆ Syntax for Upper Bound

```
class ClassName<T extends SuperClass> { }
```



Here, **T** can be **SuperClass** or any subclass of **SuperClass**.

◆ Syntax for Lower Bound

```
className<? super SubClass>
```



Here, **?** can be **SubClass** or any superclass of **SubClass**.

Upper Bounded Wildcards (extends Keyword)

Upper bounds restrict the type parameter to a specific class and its subclasses.

Example 1: Restricting Type to Number or Subclasses

```
class Calculator<T extends Number> {
    private T num;

    public Calculator(T num) {
        this.num = num;
    }

    public double getDoubleValue() {
        return num.doubleValue();
    }
}

public class UpperBoundExample {
    public static void main(String[] args) {
        Calculator<Integer> intCalc = new Calculator<>(10);
        Calculator<Double> doubleCalc = new Calculator<>(10.5);
```



```
        System.out.println("Integer value as double: " + intCalc.getDoubleVal());
        System.out.println("Double value: " + doubleCalc.getDoubleValue());
    }
}
```

- This ensures that **Calculator** only accepts **Number** types (**Integer**, **Double**, **Float**, etc.).

Lower Bounded Wildcards (super Keyword)

Lower bounds restrict the type parameter to a specific class and its subclasses.

- ◆ Useful when we want to ensure the type is of a certain **minimum type**.

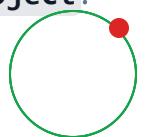
Example 2: Restricting Type to Integer or Its Superclasses

```
import java.util.List;

public class LowerBoundExample {
    public static void addNumbers(List<? super Integer> list) {
        list.add(10);
        list.add(20);
    }

    public static void main(String[] args) {
        List<Number> numbers = new java.util.ArrayList<>();
        addNumbers(numbers);
        System.out.println(numbers);
    }
}
```

- This ensures we can add **Integer** values while keeping flexibility for **Number** or **Object**.





 Pay Day Sale is Now Live!  Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



Generic Methods in Java

Introduction

In the previous blog, we discussed **Bounds on Generics**, which allow us to restrict the types of parameters in generic classes. But what if we only want to make a **single method** generic instead of an entire class? This is where **Generic Methods** come into play.

In this blog, we will cover:

- What are Generic Methods?
- Defining a Generic Method
- Using Multiple Type Parameters
- Upper and Lower Bounds in Generic Methods
- Wildcards in Generic Methods
- Generic Method vs Generic Class
- Example Use Cases

What are Generic Methods?



A **Generic Method** is a method that has its own type parameter, independent of any class-level generics. This allows a method to work with **different data types** while still maintaining **type safety**.

- ◆ **Syntax of a Generic Method:**

```
<type_parameter> returnType methodName(parameters) {  
    // method body  
}
```



- `<T>` before the return type indicates a generic method.
- `T` is the type parameter that the method will use.

Defining a Generic Method

Let's create a **simple generic method** that prints any type of array:

```
public class GenericMethodExample {  
  
    public static <T> void printArray(T[] elements) {  
        for (T element : elements) {  
            System.out.print(element + " ");  
        }  
        System.out.println();  
    }  
  
    public static void main(String[] args) {  
        Integer[] intArray = {1, 2, 3, 4, 5};  
        String[] strArray = {"Hello", "Java", "Generics"};  
  
        System.out.println("Integer Array:");  
        printArray(intArray);  
  
        System.out.println("String Array:");
```



```
    printArray(strArray);
}
```

Output:

```
Integer Array:  
1 2 3 4 5  
String Array:  
Hello Java Generics
```

- Here, the `printArray` method works with both `Integer[]` and `String[]`, demonstrating the power of generics.

Using Multiple Type Parameters

A generic method can have **multiple type parameters**.

Example: Swapping Two Elements of Any Type

```
public class SwapExample {  
  
    public static <T, U> void swap(T first, U second) {  
        System.out.println("Before Swap: " + first + " and " + second);  
  
        // Swap Logic using a temporary variable  
        U temp = second;  
        second = (U) first;  
        first = (T) temp;  
  
        System.out.println("After Swap: " + first + " and " + second);  
    }  
  
    public static void main(String[] args) {  
        swap(10, "Java");  
    }  
}
```

```
}
```

- This example shows multiple generic types (`T` and `U`), allowing the method to work with different data types.

Upper and Lower Bounds in Generic Methods

Sometimes, we may want to restrict the types of values that can be passed to a generic method. This is done using **Upper Bounds** (`extends`) and **Lower Bounds** (`super`).

Upper Bound (`extends`)

The `extends` keyword is used to **restrict** a generic type to a specific class or its subclasses.

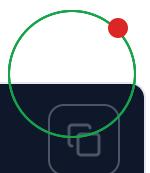
```
public class UpperBoundExample {  
  
    public static <T extends Number> void showNumber(T num) {  
        System.out.println("Number: " + num);  
    }  
  
    public static void main(String[] args) {  
        showNumber(10);           // Works with Integer  
        showNumber(5.5);          // Works with Double  
        // showNumber("Java");   // Compile-time error  
    }  
}
```



- Only `Number` and its subclasses (`Integer`, `Double`, `Float`, etc.) are allowed.

Lower Bound (`super`)

The `super` keyword restricts a generic type to a specific class or its **superclasses**.



```
import java.util.List;

public class LowerBoundExample {

    public static void addNumber(List<? super Integer> list) {
        list.add(10); // ✓ Allowed
        list.add(20); // ✓ Allowed
        // list.add(3.5); // ✗ Compile-time error
    }
}
```

- The method can accept **Integer** or any of its superclasses (**Number, Object, etc.**).

Wildcards in Generic Methods

Wildcards (?) provide **flexibility** by allowing **unknown types**.

Example: Using ? to Print Any List

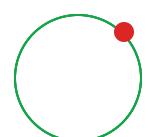
```
import java.util.List;

public class WildcardExample {

    public static void printList(List<?> list) {
        for (Object obj : list) {
            System.out.print(obj + " ");
        }
        System.out.println();
    }
}
```

- The method can accept **any type of List** (**List, List, etc.**).

Generic Method vs Generic Class



Feature	Generic Class	Generic Method
Scope	The whole class is generic	Only a specific method is generic
Type Parameters	Defined at class level	Defined within the method
Usage	Useful when multiple methods need generics	Useful when only one method needs generics
Example	<code>class Box<T></code>	<code><T> void method(T param)</code>

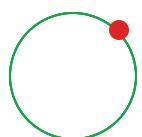
Example Use Cases of Generic Methods

1. **Utility Methods** – Printing arrays, finding maximum/minimum values.
2. **Sorting Algorithms** – Implementing generic sorting functions.
3. **Data Conversion** – Converting objects from one type to another.
4. **Wildcards & Bounds** – Providing flexible methods with constraints.

Conclusion

In this blog, we learned:

- What **Generic Methods** are and how they differ from **Generic Classes**.
- How to define and use **generic methods** with single and multiple type parameters.
- The concepts of **Upper Bounds** (`extends`), **Lower Bounds** (`super`), and **Wildcards** (`?`) in generic methods.
- Practical use cases of generic methods in real-world programming.



Java Tutorial – A Comprehensive Guide for Beginners

Introduction

Java is one of the most widely used and versatile programming languages in the world. From desktop applications to web services and mobile apps to enterprise systems, Java powers countless technologies and platforms.

This Java tutorial is designed for beginners who want to build a strong foundation in Java programming. Whether you're completely new to programming or transitioning from another language, this guide covers everything from core concepts to essential object-oriented principles and beyond.

If you're aiming to become a proficient Java developer, follow this structured path to master Java, step by step.

Prerequisites

Before you dive into Java, it's helpful (but not mandatory) to be familiar with:

- **Basic Computer Knowledge:** Understanding how software runs, what an IDE is, etc.
- **Logic Building:** Basic problem-solving and logical thinking.



- **Understanding of Programming Concepts** (optional): Knowing variables, loops, or functions from another language is a bonus.

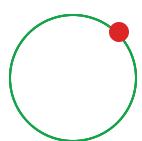
Master These Topics

Once you're ready, follow this structured list to learn Java from the ground up:

1. Introduction to Java
2. Java Fundamentals
3. Java Flow Control
4. Java Arrays
5. Java Methods
6. Java Object-Oriented Programming
7. Java Inner Classes
8. Static and Final Keywords
9. Java Exception Handling
10. Java Generics
11. Java Lambda Expressions
12. Java Collection Framework
13. Java Input/Output Streams
14. Java Reader/Writer

Introduction to Java

1. [Introduction to Java](#)
2. [Installing JDK on your computer](#)



[3. Java Comments](#)

Java Fundamentals

- [1. Java Variables and Literals](#)
- [2. Data Types in Java](#)
- [3. Operators in Java](#)
- [4. Java Basic Input and Output](#)
- [5. Java Expressions, Statements and Blocks](#)

Java Flow Control

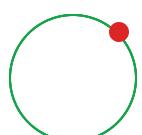
- [1. Java if else statement](#)
- [2. Java Ternary Operator](#)
- [3. Java For Loop](#)
- [4. Java while and do while loop](#)
- [5. Java continue and break statement](#)
- [6. Java Switch statement](#)

Java Arrays

- [1. Java Arrays](#)
- [2. Java Multidimensional Arrays](#)
- [3. Java Copy Arrays](#)

Java Methods

- [1. Methods in Java](#)



2. Parameter Passing in Java
3. Method Overloading
4. Variable Arguments
5. Java CommandLine Arguments
6. Java Recursive Method

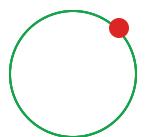
Java Object-Oriented Programming

1. Principles of Object-Oriented Programming
2. Java Class and Objects
3. Java Constructor
4. Encapsulation and Data Hiding
5. Inheritance in Java
6. Constructors in Inheritance
7. this Vs super Keyword
8. Method Overriding
9. Dynamic Method Dispatch
10. Polymorphism using Overloading and Overriding
11. Abstract Classes
12. Interfaces in Java

Java Inner Classes

1. Inner Classes in Java

Static and Final Keywords



1. Static Members and Static Blocks in Java
2. Final Members
3. Singleton Class

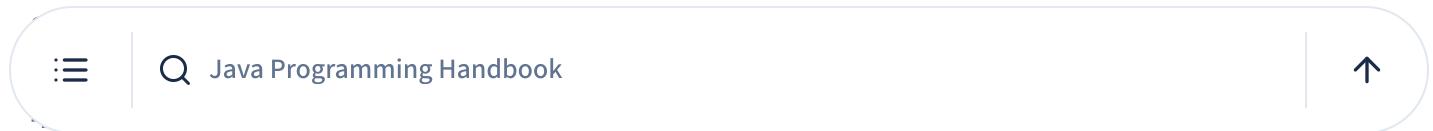
Java Exception Handling

1. What are Exceptions
2. How to handle Exception
3. Try and Catch Block
4. Multiple and Nested Try Catch
5. Class Exception
6. Checked and Unchecked Exceptions
7. Throw Vs Throws
8. Finally Block
9. Try with Resources

Java Generics

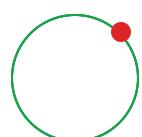


Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Lambda Expressions

1. Introduction to Lambda Expressions
2. Parameters in Lambda Expressions



3. Capture in Lambda Expression

4. Method References

Java Collection Framework

1. Collections in Java

2. Understanding Collection Interface

3. Understanding List Interface in Java

4. Java ArrayList

5. Java LinkedList

6. Java Vector

7. Java Stack

8. Understanding Queue Interface in Java

9. Java PriorityQueue

10. Java ArrayDeque

11. Understanding Set Interface in Java

12. Java HashSet

13. Java LinkedHashSet

14. Java TreeSet

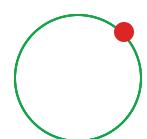
15. Understanding Map Interface in Java

16. Java HashMap

17. Java LinkedHashMap

18. Java TreeMap

19. Comparable Interface in Java



20. Comparator Interface in Java

Java Input/Output Streams

1. [What are Streams](#)
2. [InputStream and OutputStream Classes](#)
3. [FileInputStream and FileOutputStream](#)
4. [ByteArrayInputStream and ByteArrayOutputStream](#)
5. [ObjectInputStream and ObjectOutputStream](#)
6. [BufferedInputStream and BufferedOutputStream](#)
7. [Java PrintStream Class](#)

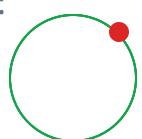
Java Reader/Writer

1. [Java File Class](#)
2. [Java Reader and Writer Class](#)
3. [InputStreamReader and OutputStreamWriter Class](#)
4. [FileReader and FileWriter Class](#)
5. [BufferedReader and BufferedWriter Class](#)
6. [StringReader and StringWriter Class](#)
7. [Java PrintWriter Class](#)

Key Features of Java

Java has remained popular for decades for good reason. Here's what makes it special:

- **Platform Independent:** Write once, run anywhere (thanks to the JVM).





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



Introduction to Lambda Expressions in Java

Introduction

Lambda expressions are one of the most powerful features introduced in Java 8. They provide a clear and concise way to represent functions, making Java code more readable and expressive. Before Java 8, anonymous inner classes were commonly used to pass behavior, but they were often verbose and cluttered. Lambda expressions solve this problem by simplifying function definitions.

In this blog, we will explore:

- What is a Lambda Expression?
- Why Use Lambda Expressions?
- Basic Syntax
- Examples of Lambda Expressions
- Functional Interfaces and Lambda Expressions

What is a Lambda Expression?

A **lambda expression** is a short block of code that takes in parameters and returns a value. It's a more compact way of implementing **functional interfaces** (interfaces with a single abstract method).



method).

Key Features:

- **Concise** – Less boilerplate code compared to anonymous inner classes.
- **Readability** – Improves code clarity and maintainability.
- **Functional Programming** – Enables a more functional style of programming in Java.

Traditional Anonymous Inner Class (Before Java 8):

```
interface Greeting {  
    void sayHello();  
}  
  
public class LambdaExample {  
    public static void main(String[] args) {  
        Greeting greeting = new Greeting() {  
            @Override  
            public void sayHello() {  
                System.out.println("Hello, World!");  
            }  
        };  
        greeting.sayHello();  
    }  
}
```

Using Lambda Expression (Java 8+):

```
interface Greeting {  
    void sayHello();  
}  
  
public class LambdaExample {  
    public static void main(String[] args) {  
        Greeting greeting = () -> System.out.println("Hello, World!");  
    }  
}
```

```
greeting.sayHello();
```

```
}
```

Notice the difference?

- No need to create an anonymous inner class.
- The `sayHello()` method implementation is directly written in a lambda expression.
- Code is cleaner and more readable.

Syntax of Lambda Expressions

Lambda expressions follow this basic syntax:

```
(parameters) -> { body }
```



Components:

1. **Parameters** – List of input parameters (optional if none are required).
2. **Arrow Token (`>`)** – Separates parameters from the function body.
3. **Body** – Defines the behavior (either a single expression or a block of statements).

Example Variations:

Lambda with No Parameters

```
() -> System.out.println("Hello, Lambda!");
```



Lambda with One Parameter



```
name -> System.out.println("Hello, " + name);
```



Lambda with Multiple Parameters

```
(a, b) -> System.out.println("Sum: " + (a + b));
```



Lambda with a Return Statement

```
(a, b) -> {  
    int sum = a + b;  
    return sum;  
};
```



If the body contains only a single statement, `{}` and `return` are optional.

Functional Interfaces and Lambda Expressions

A functional interface is an interface that contains **only one abstract method**. Lambda expressions work with functional interfaces because they provide an implementation for this single method.

Example: Functional Interface with Lambda

```
@FunctionalInterface  
interface MathOperation {  
    int operate(int a, int b);  
}  
  
public class LambdaExample {  
    public static void main(String[] args) {  
        MathOperation add = (a, b) -> a + b;  
        System.out.println("Sum: " + add.operate(10, 20));  
    }  
}
```



```
}
```

Key Takeaways:

- **MathOperation** is a functional interface (has one abstract method **operate**).
- The lambda expression **(a, b) -> a + b** provides an implementation for the **operate** method.
- The function is used like any other method.

Advantages of Using Lambda Expressions

- Reduces Boilerplate Code – No need for anonymous inner classes.
- Improves Readability – Shorter and more expressive code.
- Enhances Functional Programming – Allows passing functions as parameters.
- Works Well with Streams & Collections – Makes data manipulation easier.

Conclusion

In this blog, we introduced **Lambda Expressions in Java** and covered:

- What lambda expressions are and why they are useful.
- Basic syntax and different variations.
- How lambda expressions work with **functional interfaces**.
- Examples showcasing how they make Java code cleaner and more efficient.

Lambda expressions open the door to a **functional programming** approach in Java and are widely used in modern development, especially in combination with Streams and Collections.



Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



Parameters in Lambda Expressions in Java

Introduction

In the previous blog, we explored the **basics of lambda expressions** — what they are, how they simplify code, and how they relate to functional interfaces. In this blog, we'll focus specifically on an important part of lambda expressions: **parameters**.

Understanding how parameters work in lambda expressions is essential because they allow you to **pass data into the lambda's behavior**, making them more flexible and powerful.

In this blog, we will cover:

- What are Parameters in Lambda Expressions?
- Syntax Rules for Parameters
- Type Inference in Parameters
- Parentheses and Single Parameters
- Multiple Parameters
- Use Cases with Parameters
- Returning Values from Lambda Expressions
- Common Mistakes and Best Practices



What Are Parameters in Lambda Expressions?

Just like methods, lambda expressions can **take input parameters**. These parameters define the **data that the lambda expression will operate on**. The number and type of parameters in a lambda expression must match the abstract method of the functional interface it implements.

Syntax Rules for Parameters

Lambda expression parameters follow this syntax:

```
(parameters) -> { body }
```



1. No Parameters

```
() -> System.out.println("No parameters here!");
```



2. Single Parameter (Without Type)

```
name -> System.out.println("Hello, " + name);
```



3. Single Parameter (With Type)

```
(String name) -> System.out.println("Hello, " + name);
```



4. Multiple Parameters

```
(a, b) -> System.out.println("Sum: " + (a + b));
```



With explicit types:



```
(int a, int b) -> System.out.println("Sum: " + (a + b));
```



Type Inference in Lambda Parameters

Java uses **type inference** to determine the parameter types in most cases. This means you don't always have to write the types explicitly — the compiler can infer them based on the target functional interface.

Example:

```
// Functional Interface
interface StringLength {
    int getLength(String str);
}

// Lambda without type
StringLength length = str -> str.length();
```



Or with explicit type:

```
StringLength length = (String str) -> str.length();
```



Real-World Use Cases with Parameters

1. Sorting a List Using Lambda with Parameters

```
List<String> names = Arrays.asList("Charlie", "Alice", "Bob");
Collections.sort(names, (s1, s2) -> s1.compareTo(s2));
System.out.println(names);
```



2. Performing Math Operations

```
@FunctionalInterface  
interface MathOperation {  
    int operate(int a, int b);  
}  
  
public class LambdaMath {  
    public static void main(String[] args) {  
        MathOperation add = (a, b) -> a + b;  
        MathOperation multiply = (a, b) -> a * b;  
  
        System.out.println("Addition: " + add.operate(10, 5));  
        System.out.println("Multiplication: " + multiply.operate(10, 5));  
    }  
}
```

3. Using Lambda with Stream API

```
List<String> words = Arrays.asList("apple", "banana", "cherry");  
  
words.forEach(word -> System.out.println(word.toUpperCase()));
```

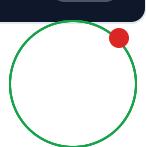
Returning Values from Lambda Expressions

Just like regular methods, lambda expressions can also **return values**. This is especially useful when you're using lambdas to implement methods that return data.

1. Single Expression (Return is Implicit)

```
(a, b) -> a + b
```

This is valid because it's a single expression. Java automatically returns the result.



2. Multiple Statements (Return is Explicit)

```
(a, b) -> {  
    int result = a + b;  
    return result;  
}
```



3. Using Return in Functional Interfaces

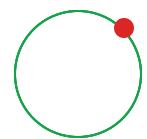
```
@FunctionalInterface  
interface StringConcat {  
    String combine(String s1, String s2);  
}  
  
public class LambdaReturn {  
    public static void main(String[] args) {  
        StringConcat concat = (s1, s2) -> s1 + " " + s2;  
        System.out.println(concat.combine("Hello", "World"));  
    }  
}
```



Parentheses in Lambda Parameters – Quick Summary

Number of Parameters	Type Specified	Parentheses Required?
0	N/A	Yes
1	No	Optional
1	Yes	Yes
2 or more	Yes/No	Always

Common Mistakes to Avoid



🚫 Missing parentheses when using type

```
// Incorrect  
StringLength length = String str -> str.length();
```



👍 Correct

```
StringLength length = (String str) -> str.length();
```



🚫 Parameter count mismatch with functional interface If the functional interface requires two parameters, your lambda must provide two.

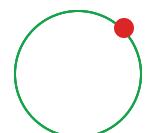
Best Practices

- Use type inference where possible to make code cleaner.
- Add parameter types explicitly only when necessary (like during complex generics or to improve readability).
- Always match the lambda parameters with the method signature of the functional interface.
- Keep lambda expressions concise and expressive. Avoid overly complex logic inside a lambda.

Conclusion

Lambda expressions become truly useful when they can accept parameters and operate on them, making your Java code concise and expressive.

In this blog, we covered:





 Pay Day Sale is Now Live!  Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



 Java Programming Handbook



Capture in Lambda Expressions

Introduction

Lambda expressions in Java can capture variables from their surrounding scope. This feature allows lambda expressions to behave similarly to anonymous inner classes, making them more powerful and flexible. Understanding how variable capture works is crucial to writing efficient and bug-free lambda-based code.

In this blog, we will cover:

- What is Variable Capture?
- Types of Variable Capture
 - Capturing Local Variables (Effectively Final Variables)
 - Capturing Instance Variables
 - Capturing Static Variables
- Examples of Variable Capture in Lambda Expressions
- Common Mistakes and Best Practices

What is Variable Capture?



Variable capture refers to the ability of a lambda expression to use variables from its surrounding scope. However, Java enforces some rules on which variables can be captured and how they can be used inside lambda expressions.

Key Rules of Variable Capture:

1. Lambda expressions can access local variables, but they must be effectively final.
2. Instance variables and static variables can be accessed freely.
3. Lambdas do not create a new scope like inner classes; they share the scope of their enclosing method.

Let's explore these types of captures in detail.

1. Capturing Local Variables (Effectively Final Variables)

Lambda expressions can capture local variables from the surrounding method, but only if they are effectively final (i.e., they do not change after initialization).

Example:

```
@FunctionalInterface  
interface Printer {  
    void print();  
}  
  
public class LambdaCaptureExample {  
    public static void main(String[] args) {  
        String message = "Hello from Lambda!"; // Effectively final  
  
        Printer printer = () -> System.out.println(message); // Capturing Loc  
        printer.print();  
    }  
}
```

Why Must Local Variables Be Effectively Final?

Java enforces this rule to prevent unpredictable behavior. If local variables were allowed to change, lambda expressions might capture outdated or inconsistent values.

Incorrect Example (Modifying a Captured Variable):

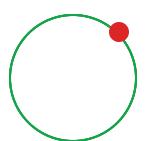
```
public class LambdaCaptureExample {  
    public static void main(String[] args) {  
        String message = "Hello";  
  
        Printer printer = () -> System.out.println(message);  
  
        message = "Hello, Lambda!"; // Compilation Error: Variable must be final  
        printer.print();  
    }  
}
```

To fix this, use an array or a wrapper class (like `AtomicReference`) to allow modifications.

Correct Example (Using an Array for Modification):

```
public class LambdaCaptureExample {  
    public static void main(String[] args) {  
        final String[] message = {"Hello"};  
  
        Printer printer = () -> System.out.println(message[0]);  
  
        message[0] = "Hello, Lambda!"; // Allowed, as array reference is final  
        printer.print();  
    }  
}
```

2. Capturing Instance Variables



Unlike local variables, **instance variables** can be captured without requiring them to be final. This is because instance variables belong to an object and are not stored on the stack like local variables.

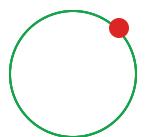
Example:

```
@FunctionalInterface  
interface Display {  
    void show();  
}  
  
public class InstanceVariableCapture {  
    private String instanceMessage = "Hello from instance variable!";  
  
    public void demonstrate() {  
        Display display = () -> System.out.println(instanceMessage);  
        display.show();  
  
        instanceMessage = "Modified instance variable!"; // Allowed  
        display.show();  
    }  
  
    public static void main(String[] args) {  
        new InstanceVariableCapture().demonstrate();  
    }  
}
```

Explanation:

- The lambda expression captures **instanceMessage**, an instance variable.
- Unlike local variables, **instanceMessage** can be modified after lambda capture.

3. Capturing Static Variables



Static variables belong to the class rather than an instance. Lambda expressions can capture static variables freely, similar to instance variables.

Example:

```
@FunctionalInterface  
interface StaticDisplay {  
    void show();  
}  
  
public class StaticVariableCapture {  
    private static String staticMessage = "Hello from static variable!";  
  
    public static void main(String[] args) {  
        StaticDisplay display = () -> System.out.println(staticMessage);  
        display.show();  
  
        staticMessage = "Modified static variable!"; // Allowed  
        display.show();  
    }  
}
```

Explanation:

- The lambda expression captures **staticMessage**, a static variable.
- Since static variables exist for the lifetime of the class, they **can be modified** after capture.

Common Mistakes and Best Practices

✖ Mistake 1: Trying to Modify Local Variables in Lambda Expressions

```
public class LambdaExample {  
    public static void main(String[] args) {  
        int number = 10;  
        Runnable task = () -> {
```



Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



Method References in Java

Introduction

Method references in Java are a shorthand notation for lambda expressions that refer to existing methods. They improve readability and make the code more concise. Instead of writing a full lambda expression, we can directly refer to a method using `:::` (double colon) operator.

In this blog, we will cover:

- What are Method References?
- Types of Method References
 - Reference to a Static Method
 - Reference to an Instance Method of a Particular Object
 - Reference to an Instance Method of an Arbitrary Object
 - Reference to a Constructor
- Examples for Each Type

What are Method References?



Method references allow us to pass a method as an argument to another method, making the code more readable and easier to understand. Instead of writing a full lambda expression, we can use a method reference when the lambda body **only calls a single existing method**.

Syntax:

```
ClassName::methodName // Static and instance methods  
objectReference::methodName // Instance methods  
ClassName::new // Constructor reference
```



Let's explore the different types of method references in detail.

1. Reference to a Static Method

This type of method reference is used when we want to refer to a static method of a class.

Example:

```
import java.util.function.Function;  
  
public class StaticMethodReference {  
    public static int square(int number) {  
        return number * number;  
    }  
  
    public static void main(String[] args) {  
        // Using Lambda expression  
        Function<Integer, Integer> lambdaSquare = (n) -> StaticMethodReference::square;  
  
        // Using method reference  
        Function<Integer, Integer> methodRefSquare = StaticMethodReference::square;  
  
        System.out.println(lambdaSquare.apply(5)); // Output: 25  
        System.out.println(methodRefSquare.apply(5)); // Output: 25  
    }  

```



When to Use: If a lambda expression directly calls a static method, use a method reference instead.

2. Reference to an Instance Method of a Particular Object

If we have an object and want to call one of its instance methods using a functional interface, we can use this method reference.

Example:

```
import java.util.function.Supplier;  
  
public class InstanceMethodReference {  
    public void sayHello() {  
        System.out.println("Hello, Method Reference!");  
    }  
  
    public static void main(String[] args) {  
        InstanceMethodReference obj = new InstanceMethodReference();  
  
        // Using Lambda expression  
        Supplier<String> lambdaGreeting = () -> obj.sayHello();  
  
        // Using method reference  
        Supplier<String> methodRefGreeting = obj::sayHello;  
  
        lambdaGreeting.get(); // Output: Hello, Method Reference!  
        methodRefGreeting.get(); // Output: Hello, Method Reference!  
    }  
}
```

When to Use: When you have an existing object and need to call its method via a functional interface.

3. Reference to an Instance Method of an Arbitrary Object

This is useful when working with **streams** or **collections**. It allows us to refer to an instance method of an unknown object (arbitrary instance of a class).

Example:

```
import java.util.Arrays;
import java.util.List;

public class ArbitraryObjectMethodReference {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

        // Using Lambda expression
        names.forEach(name -> System.out.println(name));

        // Using method reference
        names.forEach(System.out::println);
    }
}
```

When to Use: When iterating over a collection and applying the same instance method to all elements.

4. Reference to a Constructor

Constructor references allow us to create new instances without using the **new** keyword explicitly in a lambda expression.

Example:

```
import java.util.function.Supplier;
```

Java Tutorial – A Comprehensive Guide for Beginners

Introduction

Java is one of the most widely used and versatile programming languages in the world. From desktop applications to web services and mobile apps to enterprise systems, Java powers countless technologies and platforms.

This Java tutorial is designed for beginners who want to build a strong foundation in Java programming. Whether you're completely new to programming or transitioning from another language, this guide covers everything from core concepts to essential object-oriented principles and beyond.

If you're aiming to become a proficient Java developer, follow this structured path to master Java, step by step.

Prerequisites

Before you dive into Java, it's helpful (but not mandatory) to be familiar with:

- **Basic Computer Knowledge:** Understanding how software runs, what an IDE is, etc.
- **Logic Building:** Basic problem-solving and logical thinking.



- **Understanding of Programming Concepts** (optional): Knowing variables, loops, or functions from another language is a bonus.

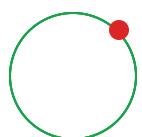
Master These Topics

Once you're ready, follow this structured list to learn Java from the ground up:

1. Introduction to Java
2. Java Fundamentals
3. Java Flow Control
4. Java Arrays
5. Java Methods
6. Java Object-Oriented Programming
7. Java Inner Classes
8. Static and Final Keywords
9. Java Exception Handling
10. Java Generics
11. Java Lambda Expressions
12. Java Collection Framework
13. Java Input/Output Streams
14. Java Reader/Writer

Introduction to Java

1. [Introduction to Java](#)
2. [Installing JDK on your computer](#)



[3. Java Comments](#)

Java Fundamentals

- [1. Java Variables and Literals](#)
- [2. Data Types in Java](#)
- [3. Operators in Java](#)
- [4. Java Basic Input and Output](#)
- [5. Java Expressions, Statements and Blocks](#)

Java Flow Control

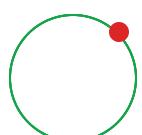
- [1. Java if else statement](#)
- [2. Java Ternary Operator](#)
- [3. Java For Loop](#)
- [4. Java while and do while loop](#)
- [5. Java continue and break statement](#)
- [6. Java Switch statement](#)

Java Arrays

- [1. Java Arrays](#)
- [2. Java Multidimensional Arrays](#)
- [3. Java Copy Arrays](#)

Java Methods

- [1. Methods in Java](#)



2. Parameter Passing in Java
3. Method Overloading
4. Variable Arguments
5. Java CommandLine Arguments
6. Java Recursive Method

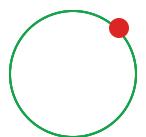
Java Object-Oriented Programming

1. Principles of Object-Oriented Programming
2. Java Class and Objects
3. Java Constructor
4. Encapsulation and Data Hiding
5. Inheritance in Java
6. Constructors in Inheritance
7. this Vs super Keyword
8. Method Overriding
9. Dynamic Method Dispatch
10. Polymorphism using Overloading and Overriding
11. Abstract Classes
12. Interfaces in Java

Java Inner Classes

1. Inner Classes in Java

Static and Final Keywords



1. Static Members and Static Blocks in Java
2. Final Members
3. Singleton Class

Java Exception Handling

1. What are Exceptions
2. How to handle Exception
3. Try and Catch Block
4. Multiple and Nested Try Catch
5. Class Exception
6. Checked and Unchecked Exceptions
7. Throw Vs Throws
8. Finally Block
9. Try with Resources

Java Generics

1. Introduction to Generics
2. Defining Generic Class
3. Bounds on Generics
4. Java Generic Methods

Java Lambda Expressions

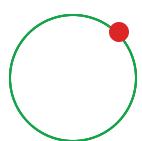


Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Collection Framework

1. [Collections in Java](#)
2. [Understanding Collection Interface](#)
3. [Understanding List Interface in Java](#)
4. [Java ArrayList](#)
5. [Java LinkedList](#)
6. [Java Vector](#)
7. [Java Stack](#)
8. [Understanding Queue Interface in Java](#)
9. [Java Priority Queue](#)
10. [Java ArrayDeque](#)
11. [Understanding Set Interface in Java](#)
12. [Java HashSet](#)
13. [Java LinkedHashSet](#)
14. [Java TreeSet](#)
15. [Understanding Map Interface in Java](#)
16. [Java HashMap](#)
17. [Java LinkedHashMap](#)
18. [Java TreeMap](#)
19. [Comparable Interface in Java](#)



20. Comparator Interface in Java

Java Input/Output Streams

1. [What are Streams](#)
2. [InputStream and OutputStream Classes](#)
3. [FileInputStream and FileOutputStream](#)
4. [ByteArrayInputStream and ByteArrayOutputStream](#)
5. [ObjectInputStream and ObjectOutputStream](#)
6. [BufferedInputStream and BufferedOutputStream](#)
7. [Java PrintStream Class](#)

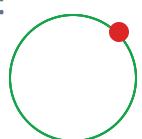
Java Reader/Writer

1. [Java File Class](#)
2. [Java Reader and Writer Class](#)
3. [InputStreamReader and OutputStreamWriter Class](#)
4. [FileReader and FileWriter Class](#)
5. [BufferedReader and BufferedWriter Class](#)
6. [StringReader and StringWriter Class](#)
7. [Java PrintWriter Class](#)

Key Features of Java

Java has remained popular for decades for good reason. Here's what makes it special:

- **Platform Independent:** Write once, run anywhere (thanks to the JVM).





 Pay Day Sale is Now Live!  Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



 Java Programming Handbook



Collections in Java

Collections in Java play a crucial role in handling and manipulating groups of objects efficiently. In this blog, we will cover everything related to Collections, including why they are needed, their implementation, and a deep dive into the Collection framework.

What is a Collection in Java?

A **Collection** in Java is a framework that provides an architecture to store and manipulate a group of objects. It allows dynamic storage, retrieval, manipulation, and communication of object elements.

Why Do We Need Collections?

Before Collections, Java used **arrays** to store multiple elements. However, arrays had limitations:

- **Fixed Size:** Once declared, the size of an array cannot be changed.
- **No Built-in Methods:** Arrays don't provide ready-made methods for searching, sorting, or modifying elements.
- **Inefficient Memory Management:** Unused space is wasted, and resizing requires creating a new array.



To overcome these issues, **Collections** were introduced, providing **dynamic size and built-in utility methods**.

Collection Framework Overview

The **Java Collection Framework (JCF)** is a hierarchy of interfaces and classes present in the **java.util** package. It provides efficient data structures for different types of collections.

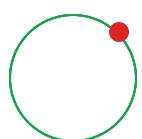
Key Features of the Collection Framework:

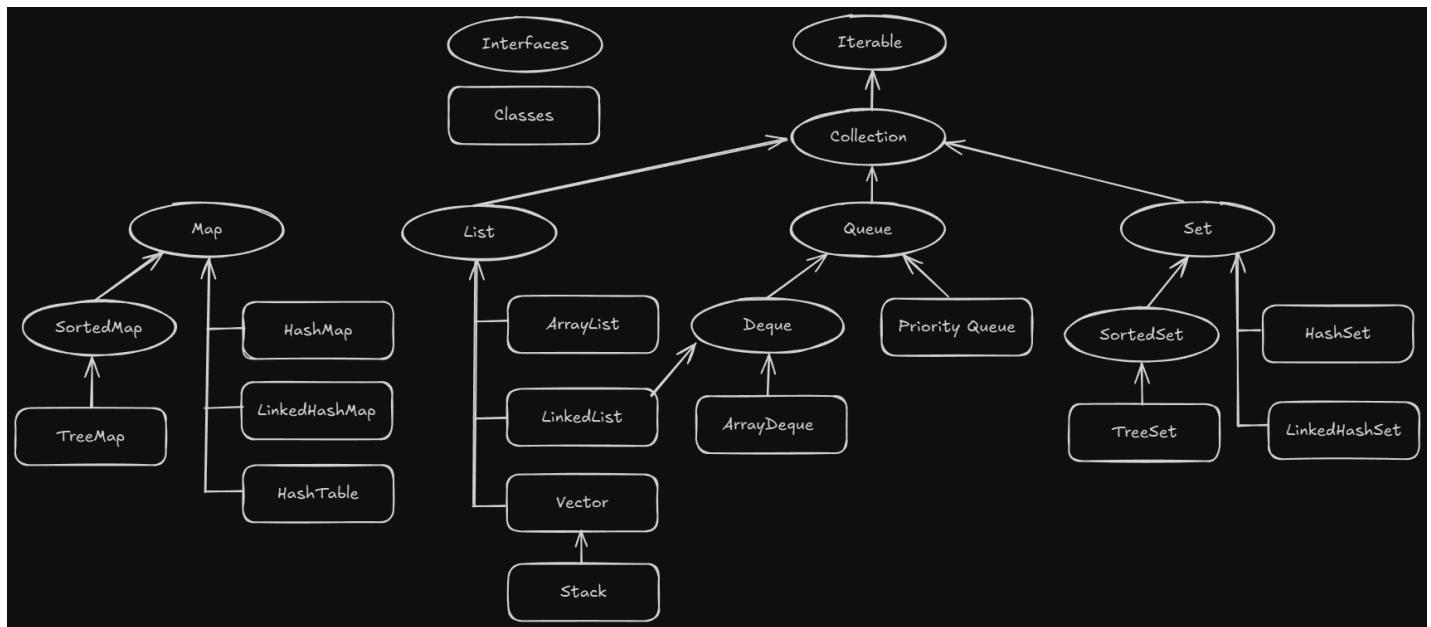
- **Reusable:** Implements standard data structures like List, Set, and Map.
- **High Performance:** Optimized implementations for speed and memory.
- **Extensible:** Allows developers to create custom implementations.

Hierarchy of Collection Framework

Java Collections are divided into three main interfaces:

- **List** (Ordered collection, allows duplicates)
- **Set** (Unordered collection, no duplicates)
- **Queue** (FIFO, used for ordered processing)
- **Map** (Key-Value pairs, not part of Collection but an important data structure)





Java Collection Framework

Why Does Collection Extend Iterable?

Collection extends **Iterable<T>**, allowing it to be iterated using an **Iterator** or an enhanced **for-each** loop.

Example:

```

import java.util.*;
public class CollectionExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("Apple");
        list.add("Banana");
        list.add("Cherry");

        for (String fruit : list) {
            System.out.println(fruit);
        }
    }
}
  
```

Output:

```
Apple  
Banana  
Cherry
```



Methods of Collection Interface

The **Collection** interface provides several useful methods:

Method	Description
<code>add(E e)</code>	Adds an element to the collection.
<code>addAll(Collection<? extends E> c)</code>	Adds all elements from another collection.
<code>remove(Object o)</code>	Removes a single instance of an element.
<code>clear()</code>	Removes all elements.
<code>contains(Object o)</code>	Checks if the collection contains a specific element.
<code>size()</code>	Returns the number of elements.
<code>isEmpty()</code>	Checks if the collection is empty.
<code>iterator()</code>	Returns an iterator for traversal.

Example:

```
import java.util.*;  
public class CollectionMethodsExample {  
    public static void main(String[] args) {  
        Collection<String> fruits = new ArrayList<>();  
        fruits.add("Mango");
```





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



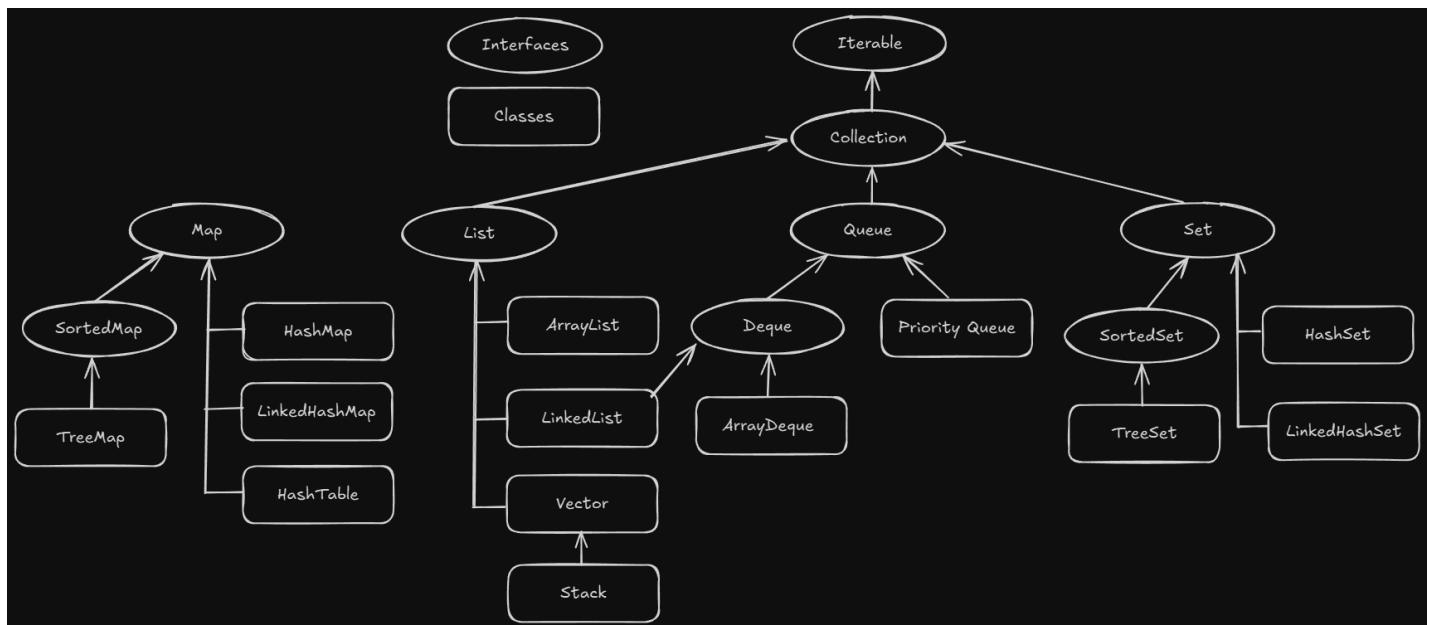
Java Programming Handbook



Understanding Collection Interface in Java

The **Collection** interface is the root interface of the Java Collection Framework and defines the fundamental methods used by all collection types. In this blog, we will explore the **Collection** interface in detail, understand its importance, and learn how to use it effectively with examples.

What is the Collection Interface?



Java Collection Interface

The **Collection** interface is part of **java.util** package and is the **base interface** for all collections like List, Set, and Queue. It extends the **Iterable<T>** interface, allowing col



to be iterated using an iterator or enhanced for-loop.

```
public interface Collection<E> extends Iterable<E> {  
    // Various collection methods  
}
```



Why is the Collection Interface Important?

- **Provides a common structure:** All collections follow a standard set of methods.
- **Encourages abstraction:** Developers can write generic code that works with any collection type.
- **Supports Polymorphism:** A method can accept `Collection<?>` as a parameter, making it flexible.

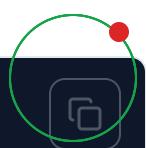
Key Methods of the Collection Interface

The `Collection` interface provides several useful methods that all collections must implement.

1. Adding Elements

Method	Description
<code>add(E e)</code>	Adds a single element to the collection.
<code>addAll(Collection<? extends E> c)</code>	Adds all elements from another collection.

Example:



```

import java.util.*;
public class CollectionAddExample {
    public static void main(String[] args) {
        Collection<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");

        Collection<String> moreFruits = Arrays.asList("Mango", "Grapes");
        fruits.addAll(moreFruits);

        System.out.println(fruits);
    }
}

```

Output:

[Apple, Banana, Mango, Grapes]



2. Removing Elements

Method	Description
<code>remove(Object o)</code>	Removes a single instance of the specified element.
<code>removeAll(Collection<?> c)</code>	Removes all elements present in another collection.
<code>clear()</code>	Removes all elements from the collection.

Example:

```

import java.util.*;
public class CollectionRemoveExample {
    public static void main(String[] args) {
        Collection<Integer> numbers = new ArrayList<>(Arrays.asList(10, 20, 3));
        numbers.remove(20);
    }
}

```



```
        System.out.println(numbers);
        numbers.clear();
        System.out.println("After clear: " + numbers);
    }
}
```

Output:

```
[10, 30, 40]
After clear: []
```



3. Checking Elements

Method	Description
<code>contains(Object o)</code>	Checks if the collection contains the specified element.
<code>containsAll(Collection<?> c)</code>	Checks if all elements of another collection exist in this collection.

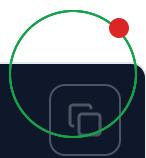
Example:

```
import java.util.*;
public class CollectionContainsExample {
    public static void main(String[] args) {
        Collection<String> cities = new ArrayList<>(Arrays.asList("New York",
                "London"));
        System.out.println("Contains London? " + cities.contains("London"));
    }
}
```



Output:

```
Contains London? true
```



4. Size and Empty Check

Method	Description
<code>size()</code>	Returns the number of elements in the collection.
<code>isEmpty()</code>	Checks if the collection is empty.

Example:

```
import java.util.*;  
public class CollectionSizeExample {  
    public static void main(String[] args) {  
        Collection<String> names = new ArrayList<>();  
  
        System.out.println("Is empty? " + names.isEmpty());  
        names.add("Alice");  
        System.out.println("Size: " + names.size());  
    }  
}
```

Output:

```
Is empty? true  
Size: 1
```

5. Iterating Over Elements

Method	Description
<code>iterator()</code>	Returns an iterator to traverse elements.

Example:



Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Understanding List Interface in Java

The **List** interface in Java is a part of the **Java Collection Framework (JCF)** and is used to store **ordered** collections of elements, allowing duplicate values. Unlike **Set**, which does not allow duplicates, **List** maintains **insertion order** and provides **positional access** to elements.

In this blog, we will explore the **List** interface, its key methods, and its implementations: **ArrayList**, **LinkedList**, **Vector**, and **Stack**.

What is the List Interface?

The **List** interface is present in the **java.util** package and extends the **Collection** interface.

```
public interface List<E> extends Collection<E> {  
    // List-specific methods  
}
```



Features of List

- **Ordered Collection:** Maintains the order in which elements are inserted.
- **Allows Duplicates:** Unlike **Set**, a **List** can store duplicate elements.
- **Index-Based Access:** Provides methods to access elements by index.



- **Dynamic Size:** Automatically resizes based on the number of elements.

Key Methods of List Interface

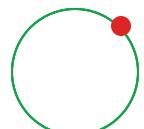
Method	Description
<code>add(E e)</code>	Adds an element to the list.
<code>add(int index, E element)</code>	Inserts an element at a specific index.
<code>get(int index)</code>	Retrieves an element at the given index.
<code>set(int index, E element)</code>	Replaces the element at the given index.
<code>remove(int index)</code>	Removes the element at the given index.
<code>indexOf(Object o)</code>	Returns the index of the first occurrence of the element.
<code>lastIndexOf(Object o)</code>	Returns the index of the last occurrence of the element.
<code>subList(int fromIndex, int toIndex)</code>	Returns a portion of the list.

Implementations of List Interface

1. ArrayList (Dynamic Array)

Characteristics:

- Uses **dynamic array** internally.
- **Fast for read operations** ($O(1)$ for get, $O(n)$ for add/remove in the middle).



- Not synchronized (not thread-safe).

Example:

```
import java.util.*;  
public class ArrayListExample {  
    public static void main(String[] args) {  
        List<String> list = new ArrayList<>();  
        list.add("Apple");  
        list.add("Banana");  
        list.add("Cherry");  
  
        System.out.println("ArrayList: " + list);  
    }  
}
```

Output:

```
ArrayList: [Apple, Banana, Cherry]
```

2. LinkedList (Doubly Linked List)

Characteristics:

- Uses **doubly linked list** internally.
- **Fast insertions & deletions** ($O(1)$ for add/remove at the start/middle, $O(n)$ for get).
- Slightly higher memory usage due to extra references.

Example:

```
import java.util.*;  
public class LinkedListExample {
```

```
public static void main(String[] args) {  
    List<Integer> list = new LinkedList<>();  
    list.add(10);  
    list.add(20);  
    list.add(30);  
  
    System.out.println("LinkedList: " + list);  
}  
}
```

Output:

```
LinkedList: [10, 20, 30]
```



3. Vector (Thread-Safe ArrayList)

Characteristics:

- Uses **dynamic array** internally (like **ArrayList**).
- **Synchronized**, making it thread-safe but slower.
- Methods are synchronized to prevent data corruption in multi-threaded environments.

Example:

```
import java.util.*;  
public class VectorExample {  
    public static void main(String[] args) {  
        List<String> list = new Vector<>();  
        list.add("Car");  
        list.add("Bike");  
        list.add("Bus");  
  
        System.out.println("Vector: " + list);  
    }  
}
```



Output:

```
Vector: [Car, Bike, Bus]
```



4. Stack (LIFO - Last In, First Out)

Characteristics:

- A subclass of **Vector** that follows LIFO (Last-In-First-Out) order.
- Provides methods like **push()**, **pop()**, **peek()** for stack operations.

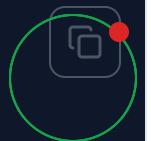
Example:

```
import java.util.*;
public class StackExample {
    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>();
        stack.push(1);
        stack.push(2);
        stack.push(3);

        System.out.println("Stack before pop: " + stack);
        System.out.println("Popped element: " + stack.pop());
        System.out.println("Stack after pop: " + stack);
    }
}
```



Output:





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Java ArrayList

The **ArrayList** class in Java is a part of the Java Collection Framework and provides a resizable-array implementation of the **List** interface. It is widely used due to its simplicity, performance, and flexibility in handling dynamic data.

1. What is an ArrayList?

- **ArrayList** is a **dynamic array**, which means it grows or shrinks as elements are added or removed.
- Unlike arrays, we don't need to specify the size while creating an **ArrayList**.

```
List<String> list = new ArrayList<>();
```



2. Internal Working

- Internally uses an array to store elements.
- When the array gets full, it creates a **new array** with a larger capacity (usually 1.5x the old size) and copies the old elements to it.

3. Constructors



```
ArrayList()           // Default constructor with initial capacity 10  
ArrayList(int initialCapacity) // Initializes with specified capacity  
ArrayList(Collection<? extends E> c) // Initializes with elements of another
```

4. Commonly Used Methods with Examples

1. add(E e) – Adds element to the end

```
ArrayList<String> list = new ArrayList<>();  
list.add("Apple");  
System.out.println(list);
```

Output:

```
[Apple]
```

2. add(int index, E element) – Inserts at specific index

```
list.add(0, "Banana");  
System.out.println(list);
```

Output:

```
[Banana, Apple]
```

3. get(int index) – Retrieves element at index

```
String item = list.get(1);  
System.out.println(item);
```

Output:

```
Apple
```



4. set(int index, E element) – Replaces element at index

```
list.set(1, "Orange");  
System.out.println(list);
```



Output:

```
[Banana, Orange]
```



5. remove(int index) – Removes element at index

```
list.remove(0);  
System.out.println(list);
```



Output:

```
[Orange]
```

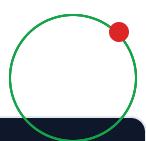


6. remove(Object o) – Removes first occurrence of element

```
list.remove("Orange");  
System.out.println(list);
```



Output:



```
[ ]
```



7. `size()` – Returns number of elements

```
System.out.println(list.size());
```



Output:

```
0
```



8. `isEmpty()` – Checks if list is empty

```
System.out.println(list.isEmpty());
```



Output:

```
true
```



9. `contains(Object o)` – Checks if element exists

```
list.add("Mango");
System.out.println(list.contains("Mango"));
```

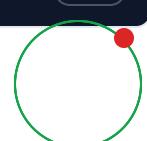


Output:

```
true
```



10. `indexOf(Object o)` – Returns first index of element



```
list.add("Banana");
System.out.println(list.indexOf("Banana"));
```



Output:

```
1
```



11. lastIndexOf(Object o) – Returns last index of element

```
list.add("Mango");
System.out.println(list.lastIndexOf("Mango"));
```



Output:

```
2
```



12. clear() – Removes all elements

```
list.clear();
System.out.println(list);
```



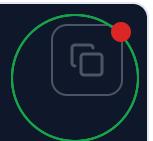
Output:

```
[]
```



13. toArray() – Converts list to array

```
ArrayList<String> fruits = new ArrayList<>();
fruits.add("Apple");
fruits.add("Banana");
```



```
String[] array = fruits.toArray(new String[0]);  
System.out.println(Arrays.toString(array));
```

Output:

```
[Apple, Banana]
```



14. `ensureCapacity(int minCapacity)` – Increases internal capacity

```
ArrayList<Integer> nums = new ArrayList<>();  
nums.ensureCapacity(100); // ensures internal capacity >= 100
```



(No visible output but improves performance if adding many elements.)

15. `trimToSize()` – Trims capacity to current size

```
nums.trimToSize(); // trims to current size
```



(No visible output, but optimizes memory usage.)

16. `forEach(Consumer<? super E> action)` – Lambda-style iteration

```
fruits.forEach(fruit -> System.out.println("Fruit: " + fruit));
```



Output:

```
Fruit: Apple  
Fruit: Banana
```



5. Performance



Operation	Time Complexity
Access (get/set)	O(1)
Add/remove at end	O(1) amortized
Add/remove middle	O(n)

6. Thread Safety

- **ArrayList** is not synchronized.
- For thread-safe operations, use:

```
List<String> syncList = Collections.synchronizedList(new ArrayList<>());
```



7. When to Use ArrayList?

- When you need fast access and iteration.
- When insertion/deletion in the middle is rare.
- When working in a single-threaded or externally synchronized environment.

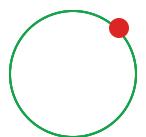
8. Best Practices

- Prefer **List** interface while declaring:

```
List<String> names = new ArrayList<>();
```



- Use **initialCapacity** if size is known beforehand.
- Avoid using **ArrayList** in multi-threaded scenarios unless synchronized.





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Java LinkedList

The **LinkedList** class in Java is part of the Java Collections Framework and implements both the **List** and **Deque** interfaces. It provides a doubly-linked list data structure.

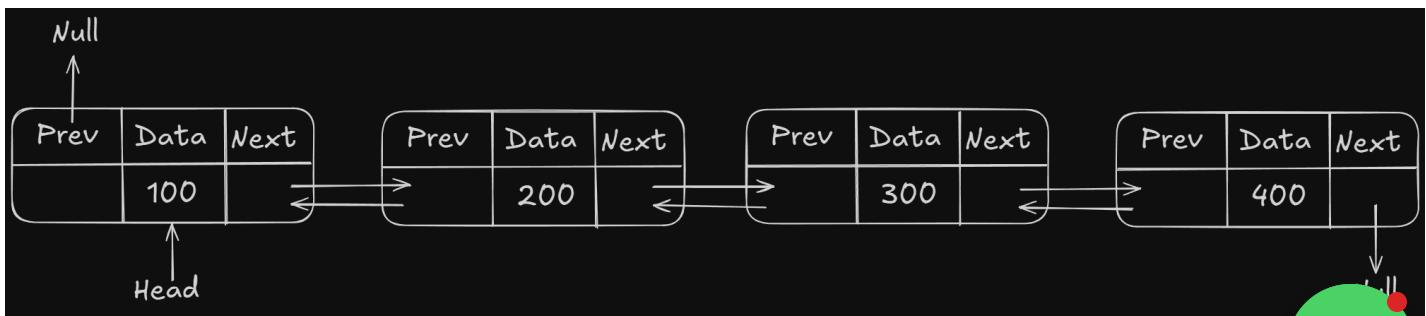
1. What is a LinkedList?

- **LinkedList** is a doubly-linked list.
- It allows elements to be added or removed from both ends.
- It can store duplicate and **null** elements.

```
List<String> list = new LinkedList<>();
```



2. Internal Working



Doubly LinkedList



- Each node contains a **data element**, a **reference to the previous node**, and a **reference to the next node**.
- Enables constant-time insertions and removals using iterators.

3. Constructors

```
LinkedList()           // Default constructor
LinkedList(Collection<? extends E>) // Constructs a List containing element
```

4. Commonly Used Methods with Examples

1. add(E e) – Adds element at the end

```
LinkedList<String> list = new LinkedList<>();
list.add("Apple");
System.out.println(list);
```

Output:

```
[Apple]
```

2. add(int index, E element) – Inserts at a specific index

```
list.add(0, "Banana");
System.out.println(list);
```

Output:

```
[Banana, Apple]
```

3. addFirst(E e) – Adds to the beginning

```
list.addFirst("Mango");
System.out.println(list);
```



Output:

```
[Mango, Banana, Apple]
```



4. addLast(E e) – Adds to the end

```
list.addLast("Grapes");
System.out.println(list);
```



Output:

```
[Mango, Banana, Apple, Grapes]
```



5. get(int index) – Retrieves element at a given index

```
String item = list.get(2);
System.out.println(item);
```



Output:

```
Apple
```



6. getFirst() – Gets the first element



```
System.out.println(list.getFirst());
```

Output:

```
Mango
```

7. getLast() – Gets the last element

```
System.out.println(list.getLast());
```

Output:

```
Grapes
```

8. remove() – Removes and returns the first element

```
list.remove();
System.out.println(list);
```

Output:

```
[Banana, Apple, Grapes]
```

9. remove(int index) – Removes element at a given index

```
list.remove(1);
System.out.println(list);
```

Output:



[Banana, Grapes]



10. `removeFirst()` – Removes the first element

```
list.removeFirst();
System.out.println(list);
```



Output:

[Grapes]



11. `removeLast()` – Removes the last element

```
list.removeLast();
System.out.println(list);
```



Output:

[]



12. `contains(Object o)` – Checks if an element exists

Output:

true



13. `size()` – Returns the number of elements in the list

```
System.out.println(list.size());
```



Output:

```
1
```



14. `isEmpty()` – Checks if the list is empty

```
System.out.println(list.isEmpty());
```



Output:

```
false
```



15. `clear()` – Removes all elements

```
list.clear();
System.out.println(list);
```



Output:

```
[]
```



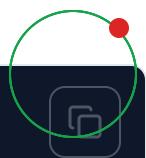
16. `offer(E e)` – Adds element at the end (queue style)

```
list.offer("Kiwi");
System.out.println(list);
```



Output:

```
[Kiwi]
```



17. peek() – Retrieves the head element without removing it

```
System.out.println(list.peek());
```



Output:

```
Kiwi
```



18. poll() – Retrieves and removes the head element

```
System.out.println(list.poll());
System.out.println(list);
```



Output:

```
Kiwi
[]
```



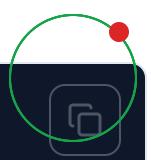
19. descendingIterator() – Iterates in reverse order

```
list.add("One");
list.add("Two");
list.add("Three");
Iterator<String> desc = list.descendingIterator();
while (desc.hasNext()) {
    System.out.print(desc.next() + " ");
}
```



Output:

```
Three Two One
```



5. Performance

Operation	Time Complexity
Access by index	$O(n)$
Add/remove at ends	$O(1)$
Add/remove in middle	$O(n)$

6. Thread Safety

- `LinkedList` is not synchronized.
- For a thread-safe version:

```
List<String> syncList = Collections.synchronizedList(new LinkedList<>());
```



7. When to Use `LinkedList`?

- When frequent insertion or removal is required.
- When random access is not a priority.
- For implementing queues, stacks, and deques.

8. Best Practices

- Prefer using the `List` interface:

```
List<String> names = new LinkedList<>();
```





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Java Vector

The **Vector** class in Java is a part of the **Java Collection Framework (JCF)** and implements the **List interface**. It uses a **dynamic array** to store the elements and is **synchronized**, making it **thread-safe**.

This blog provides a detailed understanding of the Vector class, its characteristics, constructors, commonly used methods, and usage examples.

Characteristics of Vector

- **Thread-safe:** All methods are synchronized.
- **Dynamic array:** Grows as elements are added.
- **Allows duplicates:** Like any List, Vector allows duplicate elements.
- **Maintains insertion order.**

Declaration

```
Vector<Type> vector = new Vector<>();
```



Constructors



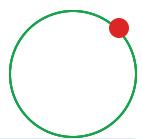
Constructor	Description
Vector()	Creates an empty vector with initial capacity of 10.
Vector(int initialCapacity)	Creates a vector with the specified initial capacity.
Vector(int initialCapacity, int capacityIncrement)	Creates a vector with specified capacity and increment.
Vector(Collection<? extends E> c)	Creates a vector containing elements of the specified collection.

Commonly Used Methods

Method	Description
add(E e)	Adds an element to the end of the vector.
add(int index, E element)	Adds an element at a specific index.
get(int index)	Returns element at specified position.
set(int index, E element)	Replaces element at specified position.
remove(int index)	Removes element at specified index.
size()	Returns the number of elements in the vector.
isEmpty()	Checks if vector is empty.
clear()	Removes all elements from the vector.

Examples

1. Adding and Printing Elements



```
import java.util.*;  
  
public class VectorAddExample {  
    public static void main(String[] args) {  
        Vector<String> vector = new Vector<>();  
        vector.add("Dog");  
        vector.add("Cat");  
        vector.add("Rabbit");  
  
        System.out.println("Vector Elements: " + vector);  
    }  
}
```

Output:

```
Vector Elements: [Dog, Cat, Rabbit]
```

2. Accessing and Modifying Elements

```
import java.util.*;  
  
public class VectorAccessExample {  
    public static void main(String[] args) {  
        Vector<Integer> vector = new Vector<>();  
        vector.add(10);  
        vector.add(20);  
        vector.add(30);  
  
        System.out.println("Element at index 1: " + vector.get(1));  
  
        vector.set(1, 25);  
        System.out.println("Modified Vector: " + vector);  
    }  
}
```

Output:

```
Element at index 1: 20  
Modified Vector: [10, 25, 30]
```

3. Removing Elements

```
import java.util.*;  
  
public class VectorRemoveExample {  
    public static void main(String[] args) {  
        Vector<String> vector = new Vector<>();  
        vector.add("Red");  
        vector.add("Green");  
        vector.add("Blue");  
  
        vector.remove(1);  
        System.out.println("After removal: " + vector);  
    }  
}
```

Output:

```
After removal: [Red, Blue]
```

4. Checking Size and Clearing Vector

```
import java.util.*;  
  
public class VectorClearExample {  
    public static void main(String[] args) {  
        Vector<Integer> vector = new Vector<>();  
        vector.add(100);  
        vector.add(200);
```





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook

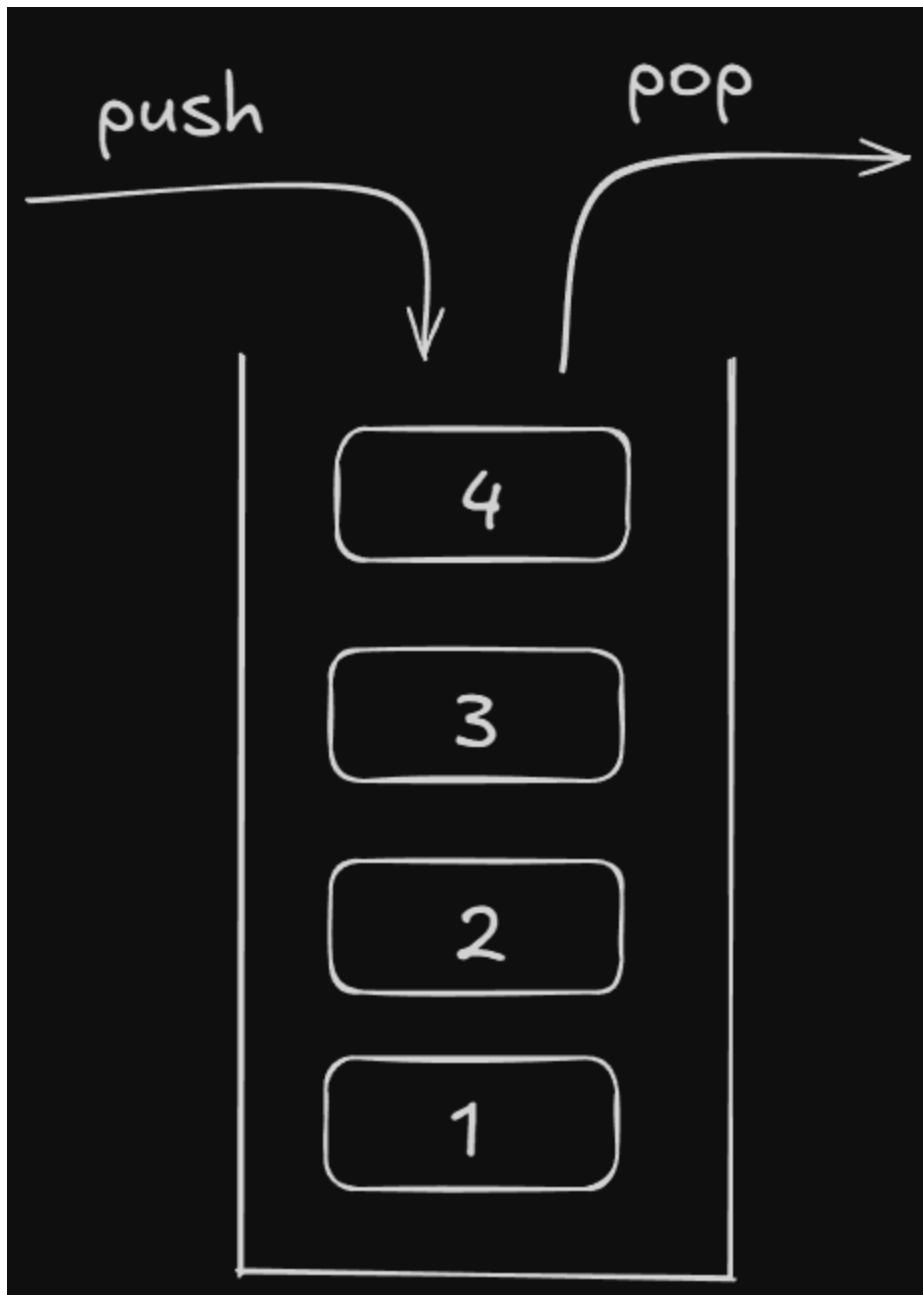


Java Stack

The **Stack** class in Java is a part of the **java.util** package and extends the **Vector** class. It represents a **Last-In-First-Out (LIFO)** data structure, where the last element added is the first one to be removed.

1. What is Stack?





Stack

- **Stack** is a subclass of **Vector**.
- It works on **LIFO** (Last in first out) principle. i.e element which is entered last will be removed first.
- It provides five key methods: **push()**, **pop()**, **peek()**, **search()**, and **empty()**.

```
Stack<Integer> stack = new Stack<>();
```



But what is Vector?

Since **Stack** extends **Vector**, it's important to understand what a **Vector** is:

- **Vector** is a growable array of objects.
- It is part of the legacy collection classes and was introduced in JDK 1.0.
- Like **ArrayList**, **Vector** uses a dynamic array to store elements.
- The key difference: **Vector** is **synchronized**, which makes it thread-safe, whereas **ArrayList** is not.

Why Stack extends Vector?

When **Stack** was introduced in Java, **Vector** was one of the only dynamic array-based structures that supported synchronization. To build on that functionality, **Stack** was created by extending **Vector** and adding LIFO-specific methods like **push()**, **pop()**, and **peek()**.

However, note that today **Deque** (like **ArrayDeque**) is often preferred due to better performance and flexibility.

2. Stack Methods with Examples

1. **push(E item)** – Adds an element to the top

```
import java.util.*;  
  
public class StackPushExample {  
    public static void main(String[] args) {  
        Stack<Integer> stack = new Stack<>();  
        stack.push(10);  
        stack.push(20);  
        stack.push(30);  
  
        System.out.println("Stack after push: " + stack);  
    }  
}
```



```
}
```

Output:

```
Stack after push: [10, 20, 30]
```



2. pop() – Removes and returns the top element

```
int popped = stack.pop();
System.out.println("Popped element: " + popped);
System.out.println("Stack after pop: " + stack);
```



Output:

```
Popped element: 30
Stack after pop: [10, 20]
```



3. peek() – Returns the top element without removing

```
int top = stack.peek();
System.out.println("Top element: " + top);
```

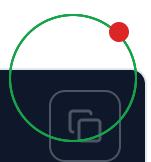


Output:

```
Top element: 20
```



4. search(Object o) – Searches the element and returns its 1-based position from the top



```
int pos = stack.search(10);
System.out.println("Position of 10 from top: " + pos);
```

Output:

```
Position of 10 from top: 2
```



5. empty() – Checks if the stack is empty

```
System.out.println("Is stack empty? " + stack.empty());
```



Output:

```
Is stack empty? false
```

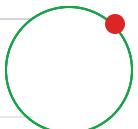


3. Use Cases of Stack

- Expression Evaluation (Postfix, Prefix)
- Undo Mechanism in editors
- Backtracking Algorithms (e.g., maze solving)
- Browser History Navigation

4. Stack vs Other List Implementations

Feature	Stack	Vector	ArrayList	LinkedList
Internal Structure	Dynamic Array	Dynamic Array	Dynamic Array	Doubly Linked List





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



Understanding Queue Interface in Java

The **Queue** interface in Java is a part of the **Java Collection Framework (JCF)** and is used to store elements in a **FIFO (First-In-First-Out)** order. This makes it suitable for scenarios like task scheduling, buffering, and managing requests in an orderly fashion.

In this blog, we will explore the **Queue** interface, its key methods, and its implementations: **PriorityQueue**, **ArrayDeque**, and **LinkedList** (as a Queue).

What is the Queue Interface?

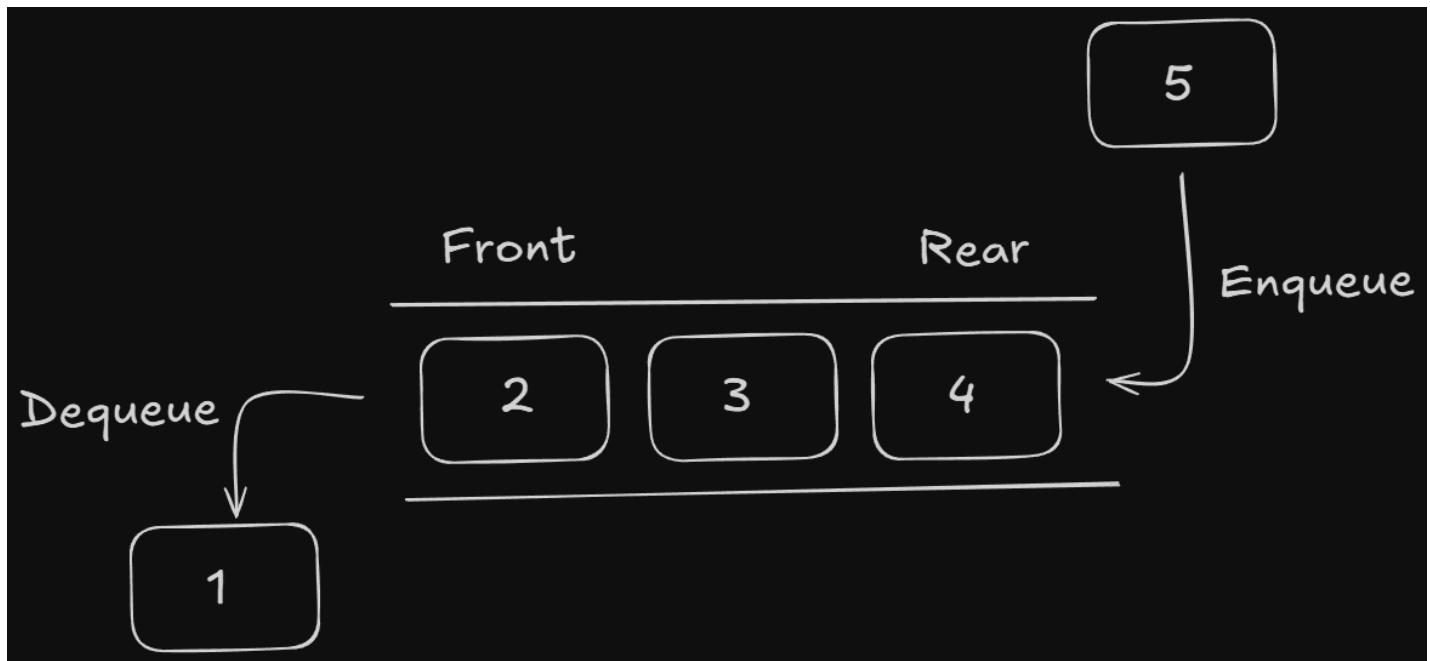
The **Queue** interface is present in the **java.util** package and extends the **Collection** interface.

```
public interface Queue<E> extends Collection<E> {  
    // Queue-specific methods  
}
```



Features of Queue



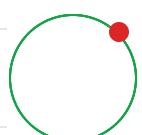


Queue in Java

- **FIFO Order:** The first element added is the first to be removed.
- **Allows Duplicates:** Unlike `Set`, a `Queue` can contain duplicate elements.
- **Dynamic Size:** Automatically resizes based on the number of elements.
- **Specialized Methods:** Includes methods for adding, removing, and inspecting elements.

Key Methods of Queue Interface

Method	Description
<code>offer(E e)</code>	Adds an element to the queue, returning <code>false</code> if it fails.
<code>add(E e)</code>	Adds an element to the queue, throwing an exception if it fails.
<code>poll()</code>	Retrieves and removes the head element, returning <code>null</code> if empty.
<code>remove()</code>	Retrieves and removes the head element, throwing an exception if empty.
<code>peek()</code>	Retrieves the head element without removing it, returning <code>null</code> if empty.



Method	Description
<code>element()</code>	Retrieves the head element without removing it, throwing an exception if empty.

Implementations of Queue Interface

1. PriorityQueue (Min Heap)

Characteristics:

- Uses a **heap** data structure internally.
- Elements are ordered based on **priority**, not insertion order.
- Does not allow `null` values.
- Not synchronized (not thread-safe).

Example:

```
import java.util.*;  
public class PriorityQueueExample {  
    public static void main(String[] args) {  
        Queue<Integer> pq = new PriorityQueue<>();  
        pq.offer(30);  
        pq.offer(10);  
        pq.offer(20);  
  
        System.out.println("PriorityQueue: " + pq);  
        System.out.println("Head element: " + pq.peek());  
    }  
}
```

Output:



PriorityQueue: [10, 30, 20]

Head element: 10



Note: The smallest element (10) is placed at the head because **PriorityQueue** uses a Min Heap.

2. ArrayDeque (Double-Ended Queue)

Characteristics:

- Uses a **resizable array** internally.
- Fast insertion & deletion from both ends.
- Allows **null values**.
- Not synchronized.

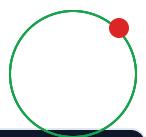
Example:

```
import java.util.*;
public class ArrayDequeExample {
    public static void main(String[] args) {
        Deque<String> deque = new ArrayDeque<>();
        deque.offer("Apple");
        deque.offer("Banana");
        deque.offerFirst("Mango");

        System.out.println("ArrayDeque: " + deque);
    }
}
```



Output:



ArrayDeque: [Mango, Apple, Banana]



3. LinkedList (As a Queue)

Characteristics:

- Implements both **Queue** and **Deque** interfaces.
- Uses a doubly linked list internally.
- Fast insertion & deletion but slower access.

Example:

```
import java.util.*;
public class LinkedListQueueExample {
    public static void main(String[] args) {
        Queue<Integer> queue = new LinkedList<>();
        queue.offer(100);
        queue.offer(200);
        queue.offer(300);

        System.out.println("LinkedList Queue: " + queue);
        System.out.println("Removed Element: " + queue.poll());
        System.out.println("Queue after poll: " + queue);
    }
}
```



Output:

```
LinkedList Queue: [100, 200, 300]
Removed Element: 100
Queue after poll: [200, 300]
```





 Pay Day Sale is Now Live!  Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



 Java Programming Handbook



Java PriorityQueue

The **PriorityQueue** in Java is part of the **Java Collections Framework** and implements the **Queue** interface. It is used when you want elements to be processed based on their priority, not just in the order they were added.

By default, **PriorityQueue** in Java functions as a **Min Heap**, meaning the smallest element is always at the head of the queue. However, we can create a Max Heap by providing a custom comparator.

Key Characteristics

- Implements **Queue** interface.
- **Heap-based** implementation.
- By default, it is a **Min Heap** (lowest value first).
- **Null elements are not allowed.**
- Not thread-safe.
- Accepts a custom **Comparator** to define ordering.

Constructors



Constructor	Description
<code>PriorityQueue()</code>	Default initial capacity (11) with natural ordering.
<code>PriorityQueue(int initialCapacity)</code>	Creates with specified capacity and natural ordering.
<code>PriorityQueue(int initialCapacity, Comparator<? super E> comparator)</code>	Creates with given capacity and ordering.
<code>PriorityQueue(Collection<? extends E> c)</code>	Creates a priority queue containing elements from the specified collection.

Commonly Used Methods (with Examples)

1. `offer(E e)` - Adds an element to the queue

```
PriorityQueue<Integer> pq = new PriorityQueue<>();
pq.offer(30);
pq.offer(10);
pq.offer(20);
System.out.println("PriorityQueue: " + pq);
```

Output:

```
PriorityQueue: [10, 30, 20]
```

2. `add(E e)` - Same as `offer`, throws exception if unable to add

```
pq.add(40);
System.out.println("After add: " + pq);
```

Output:

```
After add: [10, 30, 20, 40]
```



3. peek() - Retrieves head without removing

```
System.out.println("Peek: " + pq.peek());
```



Output:

```
Peek: 10
```



4. element() - Similar to peek, but throws exception if empty

```
System.out.println("Element: " + pq.element());
```



Output:

```
Element: 10
```



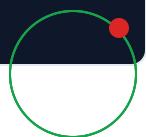
5. poll() - Retrieves and removes head, returns null if empty

```
System.out.println("Polled: " + pq.poll());
System.out.println("After poll: " + pq);
```



Output:

```
Polled: 10
After poll: [20, 30, 40]
```



6. remove() - Retrieves and removes head, throws exception if empty

```
System.out.println("Removed: " + pq.remove());  
System.out.println("After remove: " + pq);
```



Output:

```
Removed: 20  
After remove: [30, 40]
```



7. size() - Returns the size of the queue

```
System.out.println("Size: " + pq.size());
```



Output:

```
Size: 2
```



8. contains(Object o) - Checks if element exists

```
System.out.println("Contains 30? " + pq.contains(30));
```



Output:

```
Contains 30? true
```



9. toArray() - Converts queue to an array



```
Object[] arr = pq.toArray();
System.out.println("Array: " + Arrays.toString(arr));
```

Output:

```
Array: [30, 40]
```



Max Heap using PriorityQueue

```
PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());
maxHeap.offer(10);
maxHeap.offer(30);
maxHeap.offer(20);
System.out.println("MaxHeap: " + maxHeap);
```

Output:

```
MaxHeap: [30, 10, 20]
```



PriorityQueue with Custom Comparator (String Length)

```
PriorityQueue<String> stringQueue = new PriorityQueue<>(Comparator.comparingInt(String::length));
stringQueue.offer("Banana");
stringQueue.offer("Apple");
stringQueue.offer("Fig");

while (!stringQueue.isEmpty()) {
    System.out.println(stringQueue.poll());
}
```



Output:

```
Fig  
Apple  
Banana
```



Performance of PriorityQueue

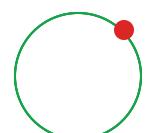
Operation	Time Complexity
<code>offer()</code> / <code>add()</code>	$O(\log n)$
<code>poll()</code> / <code>remove()</code>	$O(\log n)$
<code>peek()</code> / <code>element()</code>	$O(1)$
<code>contains()</code>	$O(n)$
Iteration	$O(n)$ (no guaranteed order)

Explanation:

- Insertion and removal are $O(\log n)$ because they involve maintaining the heap property.
- `peek()` is $O(1)$ because it just accesses the head.
- `contains()` and iteration are linear because the heap is not sorted.

Limitations of PriorityQueue

- Cannot add `null` elements.
- No constant time retrieval of arbitrary elements.
- Iteration does **not** guarantee sorted order.





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



Java ArrayDeque

The **ArrayDeque** class in Java is a **resizable-array implementation of the Deque interface**.

Unlike regular queues or stacks, **ArrayDeque** supports **insertion and removal at both ends**, making it ideal for **queue** and **stack** operations.

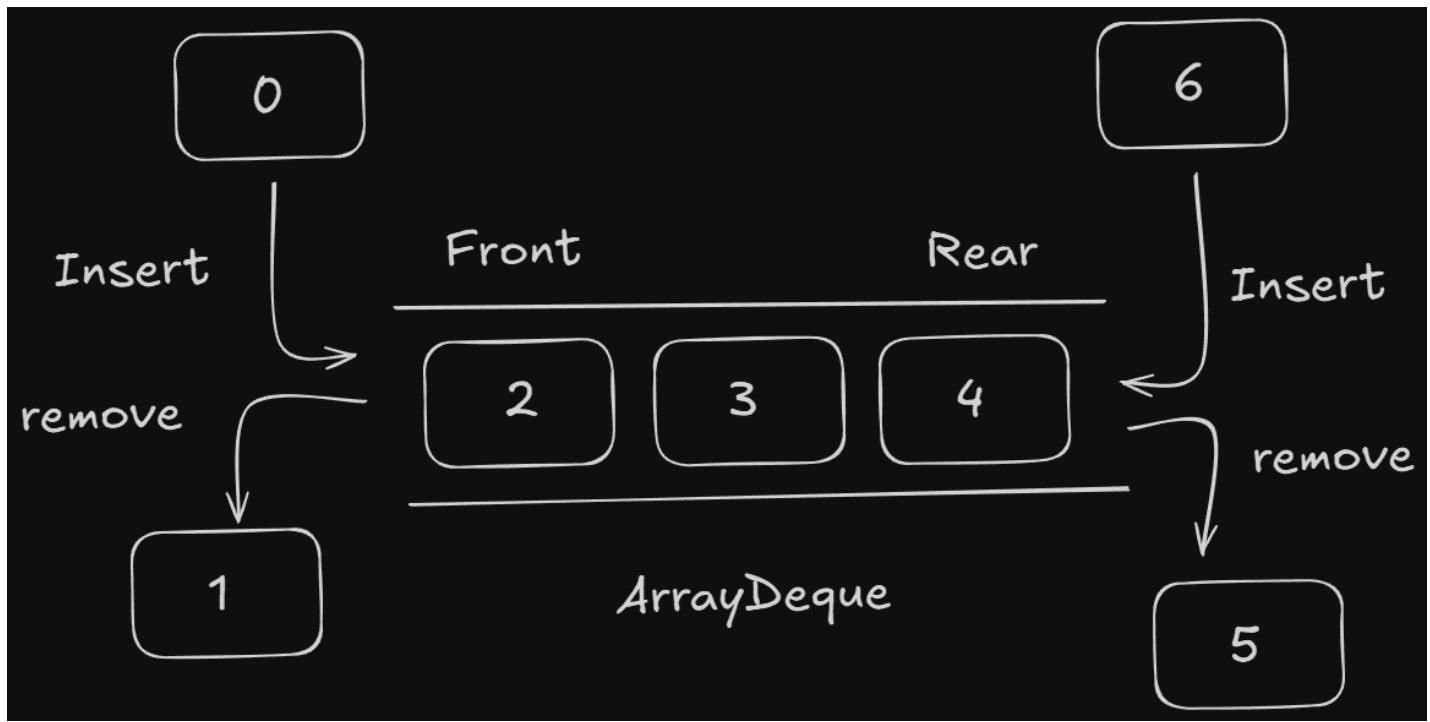
It is faster than **LinkedList** for most queue operations and does **not allow null elements**.

Key Characteristics

- Implements **Deque**, **Queue**, and **Iterable** interfaces.
- Backed by a **resizable array**.
- Faster than **Stack** and **LinkedList** for stack/queue operations.
- **Null elements are not allowed**.
- Not thread-safe (must use externally synchronized if needed).

Internal Working





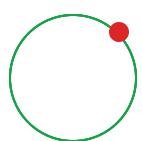
Array Deque

- Backed by a **circular array**.
- When the array is full, it resizes (doubles capacity).
- Offers **amortized constant time** for insertions/removals from both ends.

Constructors

Constructor	Description
<code>ArrayDeque()</code>	Creates an empty deque with default capacity.
<code>ArrayDeque(int numElements)</code>	Creates a deque with enough space for the given number of elements.
<code>ArrayDeque(Collection<? extends E> c)</code>	Creates a deque initialized with elements from the specified collection.

Commonly Used Methods (with Examples)



1. addFirst(E e) - Inserts element at the front

```
ArrayDeque<Integer> deque = new ArrayDeque<>();  
deque.addFirst(10);  
deque.addFirst(20);  
System.out.println("Deque: " + deque);
```

Output:

```
Deque: [20, 10]
```

2. addLast(E e) - Inserts element at the end

```
deque.addLast(30);  
System.out.println("After addLast: " + deque);
```

Output:

```
After addLast: [20, 10, 30]
```

3. offerFirst(E e) - Inserts at front, returns false if capacity restricted

```
deque.offerFirst(40);  
System.out.println("After offerFirst: " + deque);
```

Output:

```
After offerFirst: [40, 20, 10, 30]
```

4. offerLast(E e) - Inserts at rear



```
deque.offerLast(50);  
System.out.println("After offerLast: " + deque);
```



Output:

```
After offerLast: [40, 20, 10, 30, 50]
```



5. removeFirst() - Removes and returns the first element

```
System.out.println("Removed First: " + deque.removeFirst());  
System.out.println("After removeFirst: " + deque);
```



Output:

```
Removed First: 40  
After removeFirst: [20, 10, 30, 50]
```



6. removeLast() - Removes and returns the last element

```
System.out.println("Removed Last: " + deque.removeLast());  
System.out.println("After removeLast: " + deque);
```



Output:

```
Removed Last: 50  
After removeLast: [20, 10, 30]
```



7. pollFirst() - Retrieves and removes first element or null if empty



```
System.out.println("Poll First: " + deque.pollFirst());
```



Output:

```
Poll First: 20
```



8. pollLast() - Retrieves and removes last element or null if empty

```
System.out.println("Poll Last: " + deque.pollLast());
```



Output:

```
Poll Last: 30
```



9. peekFirst() - Retrieves but doesn't remove the first element

```
System.out.println("Peek First: " + deque.peekFirst());
```



Output:

```
Peek First: 10
```

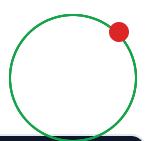


10. peekLast() - Retrieves but doesn't remove the last element

```
System.out.println("Peek Last: " + deque.peekLast());
```



Output:



Peek Last: 10



11. contains(Object o) - Checks if deque contains the element

```
System.out.println("Contains 10? " + deque.contains(10));
```



Output:

```
Contains 10? true
```



12. size() - Returns the number of elements

```
System.out.println("Size: " + deque.size());
```



Output:

```
Size: 1
```



13. clear() - Removes all elements

```
deque.clear();
System.out.println("After clear: " + deque);
```

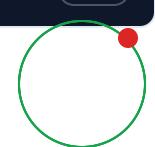


Output:

```
After clear: []
```



Performance Overview



Operation	Time Complexity
<code>addFirst() / addLast()</code>	$O(1)$ amortized
<code>removeFirst() / removeLast()</code>	$O(1)$ amortized
<code>peekFirst() / peekLast()</code>	$O(1)$
<code>contains()</code>	$O(n)$
<code>clear()</code>	$O(n)$

- Fast queue and stack operations.
- No overhead of node pointers (as in `LinkedList`).

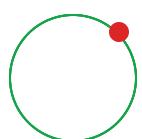
Limitations

- Not thread-safe.
- No capacity limit enforcement.
- Cannot store `null` elements (throws `NullPointerException`).

Real-World Use Cases

- Browser history (forward/backward navigation).
- Undo/Redo functionality.
- Task Scheduling (round-robin execution).
- Palindrome checking (by comparing ends).

Conclusion





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Understanding Set Interface in Java

The **Set** interface in Java is part of the **Java Collection Framework (JCF)** and is used to store a collection of unique elements. Unlike lists, sets do **not allow duplicate elements**. Sets are commonly used when you need to **eliminate duplicates** or check membership efficiently.

In this blog, we will explore the **Set** interface, its key methods, and its implementations: **HashSet**, **LinkedHashSet**, **TreeSet**, and **SortedSet**.

What is the Set Interface?

The **Set** interface is present in the **java.util** package and extends the **Collection** interface.

```
public interface Set<E> extends Collection<E> {  
    // No new methods, inherits methods from Collection  
}
```



Key Features of Set

- No Duplicate Elements:** A set cannot contain duplicate values.
- No Guarantee of Order** (depends on the implementation).
- Efficient Membership Tests:** Checking whether an element exists is faster compared to lists.



- Implements Collection Interface but adds the constraint of uniqueness.

Key Methods of Set Interface

Since **Set** extends **Collection**, it inherits its methods. Some commonly used ones include:

Method	Description
<code>add(E e)</code>	Adds an element to the set. Returns <code>false</code> if the element already exists.
<code>remove(Object o)</code>	Removes the specified element from the set.
<code>contains(Object o)</code>	Checks if the set contains the specified element.
<code>size()</code>	Returns the number of elements in the set.
<code>clear()</code>	Removes all elements from the set.

Implementations of Set Interface

1. HashSet (Unordered & Fast)

Characteristics:

- Uses a **HashTable** internally.
- No guarantee of insertion order.
- Allows `null` values.
- Fast access ($O(1)$ time complexity for `add`, `remove`, and `contains`).

Example:



```
import java.util.*;
public class HashSetExample {
    public static void main(String[] args) {
        Set<String> set = new HashSet<>();
        set.add("Apple");
        set.add("Banana");
        set.add("Mango");
        set.add("Banana"); // Duplicate, will be ignored

        System.out.println("HashSet: " + set);
    }
}
```

Output:

```
HashSet: [Apple, Banana, Mango]
```

Note: Order may vary since **HashSet** does not maintain insertion order.

2. LinkedHashSet (Ordered & Fast)

Characteristics:

- Extends **HashSet** but maintains insertion order.
- Uses a **doubly linked list** alongside a **HashTable**.
- Faster than **TreeSet** but slower than **HashSet**.

Example:

```
import java.util.*;
public class LinkedHashSetExample {
    public static void main(String[] args) {
        Set<String> set = new LinkedHashSet<>();
```

```
        set.add("Apple");
        set.add("Banana");
        set.add("Mango");

        System.out.println("LinkedHashSet: " + set);
    }
}
```

Output:

```
LinkedHashSet: [Apple, Banana, Mango]
```



Note: The order of elements remains the same as insertion order.

3. TreeSet (Sorted Order)

Characteristics:

- Implements **SortedSet** interface.
- Maintains elements in sorted (ascending) order.
- Uses a Red-Black Tree (Self-balancing BST).
- Slower than HashSet but allows sorted retrieval.

Example:

```
import java.util.*;
public class TreeSetExample {
    public static void main(String[] args) {
        Set<Integer> set = new TreeSet<>();
        set.add(30);
        set.add(10);
        set.add(20);
```



```
        System.out.println("TreeSet: " + set);
    }
```

Output:

```
TreeSet: [10, 20, 30]
```



Note: Elements are sorted in ascending order.

4. SortedSet Interface (Extended Functionality of Set)

Additional Methods in SortedSet

Method	Description
<code>first()</code>	Returns the first (lowest) element in the set.
<code>last()</code>	Returns the last (highest) element in the set.
<code>headSet(E toElement)</code>	Returns elements strictly less than <code>toElement</code> .
<code>tailSet(E fromElement)</code>	Returns elements greater than or equal to <code>fromElement</code> .

Example:

```
import java.util.*;
public class SortedSetExample {
    public static void main(String[] args) {
        SortedSet<Integer> set = new TreeSet<>();
        set.add(50);
        set.add(20);
        set.add(30);
```



```

        System.out.println("First Element: " + set.first());
        System.out.println("Last Element: " + set.last());
    }
}

```

Output:

First Element: 20
Last Element: 50



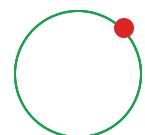
Choosing the Right Set Implementation

Feature	HashSet	LinkedHashSet	TreeSet
Order Maintained?	No	Yes (Insertion Order)	Yes (Sorted Order)
Speed (Add/Delete/Search)	Fastest	Fast	Slowest (Balanced Tree)
Allows <code>null</code> ?	Yes	Yes	No
Internal Structure	HashTable	HashTable + Linked List	Red-Black Tree
Duplicate Handling	Not Allowed	Not Allowed	Not Allowed

Conclusion

In this blog, we explored the `Set` interface and its implementations: `HashSet`, `LinkedHashSet`, `TreeSet`, and `SortedSet`.

- Use `HashSet` when fast lookup is required, and ordering is not important.
- Use `LinkedHashSet` when maintaining insertion order is needed.
- Use `TreeSet` when sorting is necessary but at the cost of performance.





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Java HashSet

HashSet is one of the most commonly used implementations of the Set interface in Java. It belongs to the **java.util** package and represents a collection that does **not allow duplicate elements**. It is backed by a **hash table**, making it highly efficient for operations like adding, removing, and checking the presence of elements.

Key Characteristics of HashSet

- **Implements Set Interface:** Inherits all characteristics of Set such as no duplicates.
- **Backed by HashMap:** Internally uses a **HashMap** to store elements.
- **No Order Guarantee:** Does not maintain the insertion order of elements.
- **Allows One null Value:** Can store a single **null** element.
- **Not Thread-safe:** Needs external synchronization for concurrent access.
- **Fast Performance:** Provides constant-time performance for basic operations like add, remove, and contains ($O(1)$ on average).

Constructors

```
HashSet<E> set = new HashSet<E>(); // Default initial capacity 16 and Load Factor 0.75  
HashSet<E> set = new HashSet<E>(int initialCapacity);
```



```
HashSet<E> set = new HashSet<>(int initialCapacity, float loadFactor);
```

Commonly Used Methods in HashSet

Method	Description
<code>add(E e)</code>	Adds the element to the set if not already present.
<code>addAll(Collection<? extends E> c)</code>	Adds all elements from another collection.
<code>remove(Object o)</code>	Removes the specified element.
<code>contains(Object o)</code>	Checks if the set contains the element.
<code>isEmpty()</code>	Returns true if the set is empty.
<code>size()</code>	Returns the number of elements in the set.
<code>clear()</code>	Removes all elements from the set.
<code>iterator()</code>	Returns an iterator over the elements.

Example: Basic Operations with HashSet

```
import java.util.*;  
  
public class HashSetExample {  
    public static void main(String[] args) {  
        HashSet<String> fruits = new HashSet<>();  
  
        fruits.add("Apple");  
        fruits.add("Banana");  
        fruits.add("Mango");  
        fruits.add("Apple"); // Duplicate, will not be added
```



```
System.out.println("HashSet: " + fruits);

System.out.println("Contains 'Banana'? " + fruits.contains("Banana"))

fruits.remove("Banana");
System.out.println("After removing 'Banana': " + fruits);

System.out.println("Size: " + fruits.size());
}

`
```

Output:

```
HashSet: [Mango, Banana, Apple]
Contains 'Banana'? true
After removing 'Banana': [Mango, Apple]
Size: 2
```

(Note: Output order may vary due to no guaranteed order in HashSet.)

Iterating over HashSet

```
HashSet<String> set = new HashSet<>();
set.add("Red");
set.add("Green");
set.add("Blue");

for (String color : set) {
    System.out.println(color);
}
```

Output:

```
Red
Green
```

(Order is not guaranteed)

Performance Analysis

Operation	Time Complexity
add	$O(1)$ average
remove	$O(1)$ average
contains	$O(1)$ average
iteration	$O(n)$

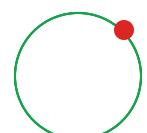
- Performance may degrade if many elements have the same hash code.
- Best suited for scenarios where quick lookup, insertion, or deletion is needed, and order does not matter.

When to Use HashSet

- When you want to store unique values.
- When you don't care about the order of elements.
- When you need high-performance operations (add, remove, contains).

Conclusion

In this blog, we explored the **HashSet** class in Java:





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



Java LinkedHashSet

The **LinkedHashSet** is a class in the Java Collection Framework that implements the **Set** interface and extends the **HashSet** class. It maintains a **linked list of the entries** in the set, preserving the **insertion order** of elements. This means that elements will be returned in the order they were inserted.

It combines **hash table performance** with **linked list ordering**, making it a great choice when you need **both uniqueness and ordering**.

Characteristics of LinkedHashSet

- Maintains **insertion order** of elements.
- **No duplicate** elements allowed.
- Allows one null element.
- Non-synchronized.
- Backed by a **hash table** with a linked list running through it.

Constructors of LinkedHashSet



Constructor	Description
<code>LinkedHashSet()</code>	Creates a new, empty set with default capacity and load factor.
<code>LinkedHashSet(int initialCapacity)</code>	Creates a new, empty set with specified initial capacity.
<code>LinkedHashSet(int initialCapacity, float loadFactor)</code>	Creates a new, empty set with specified initial capacity and load factor.
<code>LinkedHashSet(Collection<? extends E> c)</code>	Creates a set containing the elements of the specified collection.

Example:

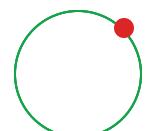
```
Set<String> set1 = new LinkedHashSet<>();
Set<Integer> set2 = new LinkedHashSet<>(50);
Set<Double> set3 = new LinkedHashSet<>(20, 0.75f);
List<String> list = Arrays.asList("A", "B", "C");
Set<String> set4 = new LinkedHashSet<>(list);
```



Commonly Used Methods

`LinkedHashSet` inherits all methods from the `Set` and `Collection` interfaces:

Method	Description
<code>add(E e)</code>	Adds the specified element if not already present.
<code>remove(Object o)</code>	Removes the specified element if present.
<code>contains(Object o)</code>	Returns true if the set contains the element.
<code>size()</code>	Returns the number of elements in the set.



Method	Description
<code>isEmpty()</code>	Returns true if the set is empty.
<code>clear()</code>	Removes all elements from the set.
<code>iterator()</code>	Returns an iterator over the elements in insertion order.

Example:

```
import java.util.*;

public class LinkedHashSetMethodsDemo {
    public static void main(String[] args) {
        Set<String> fruits = new LinkedHashSet<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Mango");
        fruits.add("Banana"); // duplicate, will be ignored

        System.out.println("Fruits: " + fruits);
        System.out.println("Contains Apple? " + fruits.contains("Apple"));
        System.out.println("Size: " + fruits.size());

        fruits.remove("Banana");
        System.out.println("After removing Banana: " + fruits);

        fruits.clear();
        System.out.println("After clear(): " + fruits);
    }
}
```

Output:

```
Fruits: [Apple, Banana, Mango]
Contains Apple? true
Size: 3
```

After removing Banana: [Apple, Mango]

Performance Analysis

Operation	Time Complexity
<code>add(E e)</code>	O(1) on average
<code>remove(Object o)</code>	O(1) on average
<code>contains(Object o)</code>	O(1) on average
<code>iteration</code>	O(n)

- Faster than TreeSet, but slightly slower than HashSet due to maintaining order.
- Ideal when both fast access and insertion order are needed.

Use Cases

- When you need to maintain insertion order.
- Removing duplicates from a list while keeping original order.
- Caching or history tracking systems.

Example: Removing Duplicates from List While Maintaining Order

```
import java.util.*;  
  
public class UniqueOrderedList {  
    public static void main(String[] args) {  
        List<String> names = Arrays.asList("Alice", "Bob", "Alice", "David",  
        Set<String> uniqueNames = new LinkedHashSet<>(names);  
    }  
}
```



Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



Java TreeSet

Introduction

In the Java Collection Framework, **TreeSet** is a class that implements the **NavigableSet** interface and indirectly implements the **SortedSet** interface, which in turn extends the **Set** interface. It stores elements in a **sorted** (natural or custom) order and is part of the **java.util** package. Internally, it is backed by a **Red-Black Tree**, a self-balancing binary search tree. Unlike **HashSet** and **LinkedHashSet**, **TreeSet** does not allow **null** elements and ensures that elements are always sorted.

Key Features

- Stores unique elements like any **Set**.
- Maintains elements in **sorted order**.
- Implements **SortedSet** and **NavigableSet** interfaces.
- Does **not allow null** elements.
- Backed by a Red-Black Tree.

Classes Implemented by TreeSet



```
public class TreeSet<E> extends AbstractSet<E>
    implements NavigableSet<E>, Cloneable, java.io.Serializable

public interface NavigableSet<E> extends SortedSet<E>
```

Here TreeSet implements NavigableSet and NavigableSet extends SortedSet, therefore TreeSet Implements SortedSet interface indirectly.

SortedSet Interface

SortedSet is a subinterface of **Set** that maintains **elements in a sorted order**.

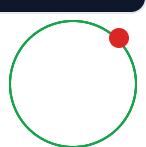
A **TreeSet** implements this interface to keep elements **automatically sorted** in natural order or using a custom comparator.

```
public interface SortedSet<E> extends Set<E> {
    Comparator<? super E> comparator();
    E first();
    E last();
    SortedSet<E> headSet(E toElement);
    SortedSet<E> tailSet(E fromElement);
    SortedSet<E> subSet(E fromElement, E toElement);
}
```

Constructors

```
TreeSet<E> set = new TreeSet<>();
TreeSet<E> set = new TreeSet<>(Comparator<? super E> comparator);
TreeSet<E> set = new TreeSet<>(Collection<? extends E> collection);
TreeSet<E> set = new TreeSet<>(SortedSet<E> sortedSet);
```

Explanation:



- The default constructor creates an empty `TreeSet` that will be sorted according to the natural ordering of its elements.
- The constructor with `Comparator` allows for custom sorting.
- The constructor with `Collection` copies elements and maintains the sorted order.
- The constructor with `SortedSet` creates a `TreeSet` with the same ordering as the provided sorted set.

Commonly Used Methods

Method	Description
<code>add(E e)</code>	Adds the specified element. Throws <code>NullPointerException</code> if element is null.
<code>remove(Object o)</code>	Removes the specified element if present.
<code>contains(Object o)</code>	Returns <code>true</code> if the element is present.
<code>isEmpty()</code>	Checks if the set is empty.
<code>size()</code>	Returns the number of elements.
<code>clear()</code>	Removes all elements.
<code>first()</code>	Returns the lowest element.
<code>last()</code>	Returns the highest element.
<code>headSet(E toElement)</code>	Returns elements strictly less than <code>toElement</code> .
<code>tailSet(E fromElement)</code>	Returns elements greater than or equal to <code>fromElement</code> .
<code>subSet(E fromElement, E toElement)</code>	Returns elements ranging from <code>fromElement</code> (inclusive) to <code>toElement</code> (exclusive).

Example: Basic Usage

```
import java.util.TreeSet;

public class TreeSetExample {
    public static void main(String[] args) {
        TreeSet<Integer> set = new TreeSet<>();
        set.add(30);
        set.add(10);
        set.add(20);
        set.add(40);

        System.out.println("TreeSet: " + set);
    }
}
```

Output:

```
TreeSet: [10, 20, 30, 40]
```

Explanation:

The numbers are automatically sorted in ascending order (natural ordering). Duplicates are not allowed, and **TreeSet** ensures sorted uniqueness.

Example: Navigable Methods

```
import java.util.TreeSet;

public class NavigableTreeSet {
    public static void main(String[] args) {
        TreeSet<Integer> set = new TreeSet<>();
        set.add(10);
        set.add(20);
```

```

set.add(30);
set.add(40);

System.out.println("First: " + set.first());
System.out.println("Last: " + set.last());
System.out.println("HeadSet(30): " + set.headSet(30));
System.out.println("TailSet(20): " + set.tailSet(20));
System.out.println("SubSet(10, 30): " + set.subSet(10, 30));
}

```

Output:

```

First: 10
Last: 40
HeadSet(30): [10, 20]
TailSet(20): [20, 30, 40]
SubSet(10, 30): [10, 20]

```

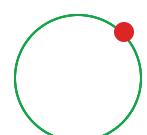


Explanation:

- `first()` returns the lowest element in the set.
- `last()` returns the highest element.
- `headSet(30)` returns elements strictly less than 30.
- `tailSet(20)` returns elements greater than or equal to 20.
- `subSet(10, 30)` returns elements from 10 (inclusive) to 30 (exclusive).

Performance

Operation	Time Complexity
<code>add(E e)</code>	$O(\log n)$



Operation Time Complexity

`remove(Object o)` $O(\log n)$

`contains(Object o)` $O(\log n)$

`iteration` $O(n)$

Explanation:

Due to the Red-Black Tree structure, insertions, deletions, and lookups are all logarithmic in time. Iterating through the set is linear.

Null Handling

```
import java.util.TreeSet;

public class NullHandling {
    public static void main(String[] args) {
        TreeSet<String> set = new TreeSet<>();
        set.add(null); // Throws NullPointerException
    }
}
```



Explanation:

`TreeSet` does not allow `null` elements because it uses comparisons to sort elements, and comparing `null` with other objects throws a `NullPointerException`.

Custom Comparator

You can sort elements in a custom order using a comparator:



```
import java.util.*;  
  
public class CustomComparatorExample {  
    public static void main(String[] args) {  
        TreeSet<String> set = new TreeSet<>(Comparator.reverseOrder());  
        set.add("Apple");  
        set.add("Banana");  
        set.add("Mango");  
  
        System.out.println("Descending Order: " + set);  
    }  
}
```

Output:

```
Descending Order: [Mango, Banana, Apple]
```

Explanation:

We passed a reverse order comparator to the `TreeSet` constructor, so the elements are sorted in descending order.

Key Takeaways:

- `TreeSet` stores **unique, sorted** elements.
- Sorting can be natural (by default) or custom (via `Comparator`).
- Null elements are **not allowed**.
- Operations like add, remove, and search take $O(\log n)$ time.

Use `TreeSet` when:

- You need sorted order and uniqueness.



Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



Understanding Map Interface in Java

The **Map** interface in Java is a part of the **Java Collection Framework (JCF)** but is **not** a part of the **Collection** interface. A **Map** is used to store key-value pairs, where each key is unique, and each key maps to **at most one value**.

In this blog, we will explore the **Map** interface, its key methods, and its implementations: **HashMap**, **LinkedHashMap**, **TreeMap**, and **Hashtable**.

What is the Map Interface?

The **Map** interface is present in the **java.util** package and is used to store data in the form of key-value pairs.

```
public interface Map<K, V> {  
    // Interface definition  
}
```



Key Features of Map

- **Stores Key-Value Pairs:** Each key is unique and maps to a single value.
- **Efficient Lookup:** Searching for a value by key is fast.
- **No Duplicate Keys:** A key can map to only one value at a time.



- Allows Null Keys and Values (depending on the implementation).

Key Methods of Map Interface

Method	Description
<code>put(K key, V value)</code>	Inserts a key-value pair into the map.
<code>get(Object key)</code>	Retrieves the value associated with a key.
<code>remove(Object key)</code>	Removes the key-value pair from the map.
<code>containsKey(Object key)</code>	Checks if a key is present in the map.
<code>containsValue(Object value)</code>	Checks if a value is present in the map.
<code>size()</code>	Returns the number of key-value pairs in the map.
<code>clear()</code>	Removes all elements from the map.

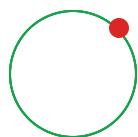
Implementations of Map Interface

1. HashMap (Unordered & Fast)

Characteristics:

- Uses a **HashTable** internally.
- No guarantee of insertion order.
- Allows one null key and multiple null values.
- Fast access ($O(1)$ time complexity for `put`, `get`, `remove`).

Example:



```
import java.util.*;
public class HashMapExample {
    public static void main(String[] args) {
        Map<Integer, String> map = new HashMap<>();
        map.put(1, "Apple");
        map.put(2, "Banana");
        map.put(3, "Mango");
        map.put(1, "Orange"); // Overwrites existing value

        System.out.println("HashMap: " + map);
    }
}
```

Output:

```
HashMap: {1=Orange, 2=Banana, 3=Mango}
```

Note: Order may vary since **HashMap** does not maintain insertion order.

2. LinkedHashMap (Ordered & Fast)

Characteristics:

- Extends **HashMap** but maintains insertion order.
- Uses a **doubly linked list** alongside a **HashTable**.
- Faster than **TreeMap** but slightly slower than **HashMap**.

Example:

```
import java.util.*;
public class LinkedHashMapExample {
    public static void main(String[] args) {
        Map<Integer, String> map = new LinkedHashMap<>();
```

```
    map.put(1, "Apple");
    map.put(2, "Banana");
    map.put(3, "Mango");

    System.out.println("LinkedHashMap: " + map);
}

}
```

Output:

```
LinkedHashMap: {1=Apple, 2=Banana, 3=Mango}
```



Note: The order of elements remains the same as insertion order.

3. TreeMap (Sorted Order)

Characteristics:

- Implements **SortedMap** interface.
- Maintains elements in sorted (ascending) order.
- Uses a Red-Black Tree (Self-balancing BST).
- Slower than HashMap but allows sorted retrieval.

Example:

```
import java.util.*;
public class TreeMapExample {
    public static void main(String[] args) {
        Map<Integer, String> map = new TreeMap<>();
        map.put(30, "Thirty");
        map.put(10, "Ten");
        map.put(20, "Twenty");
```



```
        System.out.println("TreeMap: " + map);
    }
```

Output:

```
TreeMap: {10=Ten, 20=Twenty, 30=Thirty}
```



Note: Elements are sorted in ascending order of keys.

4. Hashtable (Thread-Safe but Slow)

Characteristics:

- Synchronized and thread-safe.
- Does not allow null keys or values.
- Slower than HashMap due to synchronization overhead.

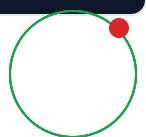
Example:

```
import java.util.*;
public class HashtableExample {
    public static void main(String[] args) {
        Map<Integer, String> map = new Hashtable<>();
        map.put(1, "Apple");
        map.put(2, "Banana");
        map.put(3, "Mango");

        System.out.println("Hashtable: " + map);
    }
}
```



Output:



Hashtable: {1=Apple, 2=Banana, 3=Mango}



Note: Unlike HashMap, it does not allow `null` keys or values.

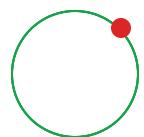
Choosing the Right Map Implementation

Feature	HashMap	LinkedHashMap	TreeMap	Hashtable
Order Maintained?	No	Yes (Insertion Order)	Yes (Sorted Order)	No
Speed (Put/Get/Delete)	Fastest	Fast	Slowest (Balanced Tree)	Slow (Thread-safe)
Allows <code>null</code> ?	Yes (1 key, multiple values)	Yes (1 key, multiple values)	No	No
Thread-Safe?	No	No	No	Yes
Internal Structure	HashTable	HashTable + Linked List	Red-Black Tree	Synchronized HashTable

Conclusion

In this blog, we explored the `Map` interface and its implementations: `HashMap`, `LinkedHashMap`, `TreeMap`, and `Hashtable`.

- Use `HashMap` when fast lookup is required, and ordering is not important.
- Use `LinkedHashMap` when maintaining insertion order is needed.
- Use `TreeMap` when sorting is necessary but at the cost of performance.
- Use `Hashtable` when thread-safety is required but at the cost of speed.





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook

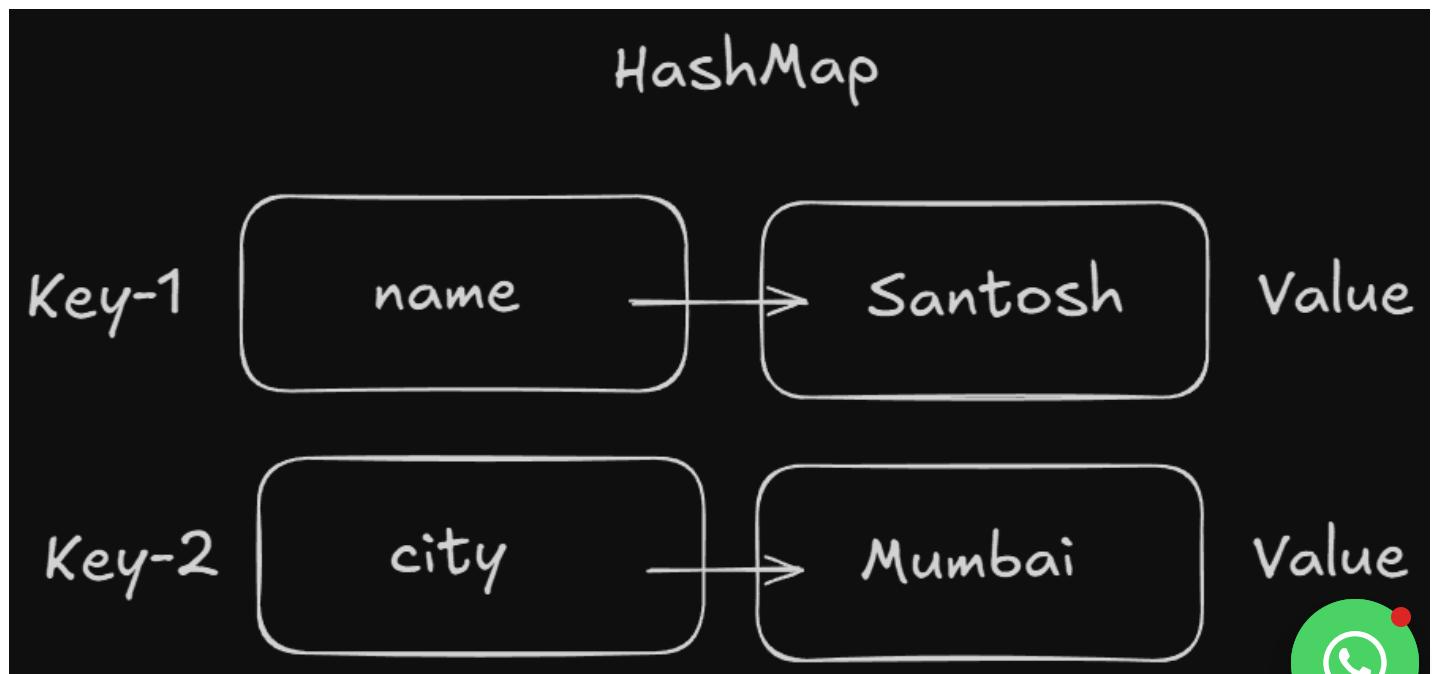


Java HashMap

Introduction

The **HashMap** class is one of the most commonly used data structures in Java, designed to store key-value pairs efficiently. It is a part of the **java.util** package and implements the **Map** interface.

This blog will guide you through everything you need to know about **HashMap**, including its working mechanism, key methods, examples, internal performance considerations, and a conclusion.



Key Characteristics

- Implements the **Map** interface.
- Stores data as key-value pairs.
- Keys must be unique.
- Allows one **null** key and multiple **null** values.
- Does **not** maintain any insertion or sorted order.
- Not synchronized (not thread-safe).

HashMap Syntax

```
HashMap<K, V> map = new HashMap<>();
```



Example:

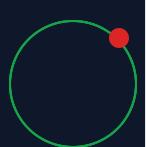
```
HashMap<Integer, String> map = new HashMap<>();
```



Commonly Used Methods in HashMap (With Examples and Output)

1. **put(K key, V value)** – Add or update key-value pair

Note:- Insertion order is not maintained here, To maintain insertion order use **LinkedHashMap** which we will see in our next blog.



```
HashMap<String, String> map = new HashMap<>();  
map.put("name", "John");  
map.put("city", "New York");  
map.put("name", "David"); // Updates the value for "name"  
System.out.println(map);
```

Output:

```
{name=David, city=New York}
```



2. get(Object key) – Retrieve value for a key

```
String value = map.get("city");  
System.out.println(value);
```



Output:

```
New York
```



3. remove(Object key) – Remove mapping by key

```
map.remove("city");  
System.out.println(map);
```

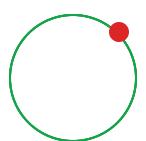


Output:

```
{name=David}
```



4. containsKey(Object key) – Check if key exists



```
boolean exists = map.containsKey("name");
System.out.println(exists);
```



Output:

```
true
```



5. `containsValue(Object value)` – Check if value exists

```
boolean hasValue = map.containsValue("David");
System.out.println(hasValue);
```



Output:

```
true
```



6. `size()` – Total number of key-value pairs

```
int count = map.size();
System.out.println(count);
```



Output:

```
1
```



7. `isEmpty()` – Check if map is empty

```
boolean empty = map.isEmpty();
System.out.println(empty);
```



Output:

```
false
```



8. `clear()` – Remove all key-value pairs

```
map.clear();
System.out.println(map);
```



Output:

```
{}
```



9. `keySet()` – Get all keys as a Set

```
map.put("name", "Alice");
map.put("city", "Delhi");
Set<String> keys = map.keySet();
System.out.println(keys);
```



Output:

```
[name, city]
```

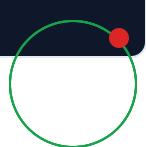


10. `values()` – Get all values as a Collection

```
Collection<String> values = map.values();
System.out.println(values);
```



Output:



[Alice, Delhi]



11. entrySet() – Get all key-value pairs as a Set of Map.Entry

```
for (Map.Entry<String, String> entry : map.entrySet()) {  
    System.out.println(entry.getKey() + " => " + entry.getValue());  
}
```



Output:

```
name => Alice  
city => Delhi
```

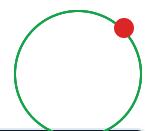


Example: Basic HashMap Usage

```
import java.util.*;  
  
public class HashMapExample {  
    public static void main(String[] args) {  
        HashMap<Integer, String> map = new HashMap<>();  
  
        map.put(1, "Apple");  
        map.put(2, "Banana");  
        map.put(3, "Mango");  
        map.put(1, "Orange"); // Overwrites value for key 1  
  
        System.out.println("HashMap: " + map);  
    }  
}
```



Output:



HashMap: {1=Orange, 2=Banana, 3=Mango}



Example: Iterating through HashMap

```
HashMap<String, Integer> map = new HashMap<>();  
map.put("Math", 90);  
map.put("Science", 85);  
map.put("English", 95);  
  
for (Map.Entry<String, Integer> entry : map.entrySet()) {  
    System.out.println(entry.getKey() + ": " + entry.getValue());  
}
```



Output:

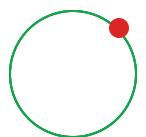
```
Math: 90  
Science: 85  
English: 95
```



Internal Working of HashMap

- Internally, HashMap uses an **array of Node objects (buckets)**.
- When you insert a key, it computes the **hash code**, which determines the bucket index.
- If multiple keys map to the same index (**collision**), the entries are stored in a **linked list** or **balanced tree** (Java 8+).

Performance and Time Complexity





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Java LinkedHashMap

In this blog, we will explore one of the commonly used implementations of the Map interface in Java: **LinkedHashMap**. We will understand its internal working, all its important methods with examples and outputs, performance aspects, and when to use it.

What is LinkedHashMap?

LinkedHashMap is a class in Java that implements the **Map** interface and extends **HashMap**. Unlike **HashMap**, it maintains the insertion order of elements.

Class Declaration

```
public class LinkedHashMap<K, V> extends HashMap<K, V> implements Map<K, V>
```

Key Characteristics

- Maintains insertion order using a doubly linked list.
- Allows one null key and multiple null values.
- Not synchronized (not thread-safe by default).
- Provides faster iteration compared to HashMap due to predictable order.



- Offers constant-time performance for basic operations like `get()` and `put()`.

Internal Working of LinkedHashMap

Internally, `LinkedHashMap` works just like `HashMap` but with an extra linked list that keeps track of the insertion order of entries.

Each node in `LinkedHashMap` contains:

- Key
- Value
- Hash
- Next node (for handling collisions)
- Before and after pointers (for doubly linked list)

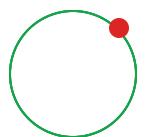
When you insert an entry:

- The key's hash is used to find the bucket (like `HashMap`).
- The entry is also linked to the previous and next entry to maintain order.

Constructors

```
LinkedHashMap<K, V> map = new LinkedHashMap<>();
LinkedHashMap<K, V> map = new LinkedHashMap<>(int initialCapacity);
LinkedHashMap<K, V> map = new LinkedHashMap<>(int initialCapacity, float load
LinkedHashMap<K, V> map = new LinkedHashMap<>(int initialCapacity, float load
```

- The last constructor maintains access order if `accessOrder` is `true`.



Important Methods with Examples

1. `put(K key, V value)`

Inserts a key-value pair.

```
LinkedHashMap<Integer, String> map = new LinkedHashMap<>();  
map.put(1, "Apple");  
map.put(2, "Banana");  
map.put(3, "Mango");  
System.out.println(map);
```

Output:

```
{1=Apple, 2=Banana, 3=Mango}
```

2. `get(Object key)`

Returns the value for the given key.

```
System.out.println(map.get(2));
```

Output:

```
Banana
```

3. `remove(Object key)`

Removes the key-value pair.

```
map.remove(1);  
System.out.println(map);
```

Output:

```
{2=Banana, 3=Mango}
```



4. containsKey(Object key)

Checks if the key exists.

```
System.out.println(map.containsKey(3));
```



Output:

```
true
```



5. containsValue(Object value)

Checks if the value exists.

```
System.out.println(map.containsValue("Banana"));
```



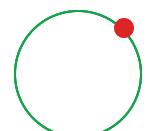
Output:

```
true
```



6. keySet()

Returns a set of keys.



```
System.out.println(map.keySet());
```



Output:

```
[2, 3]
```



7. values()

Returns a collection of values.

```
System.out.println(map.values());
```



Output:

```
[Banana, Mango]
```



8. entrySet()

Returns a set of key-value pairs.

```
System.out.println(map.entrySet());
```



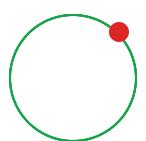
Output:

```
[ 2=Banana, 3=Mango]
```



9. clear()

Removes all entries from the map.



```
map.clear();
System.out.println(map);
```



Output:

```
{}
```



Performance of LinkedHashMap

- Basic operations like `get()`, `put()`, `remove()` are O(1) on average.
- Slightly slower than `HashMap` due to the overhead of maintaining the insertion order.
- Provides predictable iteration, which is useful when order matters.

When to Use LinkedHashMap

- When you need **fast lookups** and also want to **maintain insertion order**.
- When you want **predictable iteration order**.
- When you're implementing an **LRU (Least Recently Used) cache** using access order.

Conclusion

In this blog, we explored the `LinkedHashMap` class in Java:

- It maintains insertion order unlike `HashMap`.
- Internally uses a **doubly linked list** along with hash buckets.
- We learned about all important methods like `put()`, `get()`, `remove()`, `keySet()`, and `entrySet()`.





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Java TreeMap

The **TreeMap** class in Java is part of the **Java Collection Framework** and implements the **Map** and **SortedMap** interfaces. It stores key-value pairs in **sorted order** based on the natural ordering of keys or by a custom comparator.

What is TreeMap?

A **TreeMap** is a Red-Black Tree-based implementation of the **NavigableMap** interface. It automatically keeps the keys in **sorted ascending order**.

It implements:

- **Map** Interface – Stores key-value pairs.
- **SortedMap** Interface – Maintains keys in sorted order.

```
public class TreeMap<K,V>
    extends AbstractMap<K,V>
    implements NavigableMap<K,V>, Cloneable, java.io.Serializable

public interface NavigableMap<K,V> extends SortedMap<K,V>
```

Here TreeMap implements NavigableMap and NavigableMap extends SortedMap, therefore TreeMap indirectly implements SortedMap interface.



SortedMap Interface

SortedMap is a subinterface of **Map** that provides additional methods for dealing with **sorted keys**. A **TreeMap** uses this interface to keep its keys **automatically sorted** in natural order (or using a custom comparator).

```
public interface SortedMap<K, V> extends Map<K, V> {  
    Comparator<? super K> comparator();  
    K firstKey();  
    K lastKey();  
    SortedMap<K, V> headMap(K toKey);  
    SortedMap<K, V> tailMap(K fromKey);  
    SortedMap<K, V> subMap(K fromKey, K toKey);  
}
```

Characteristics of TreeMap:

- Stores data as **key-value pairs**.
- Maintains keys in **ascending sorted order**.
- Does not allow **null keys** (throws **NullPointerException**).
- Allows **multiple null values**.
- Not **thread-safe** (must be synchronized externally if used in multi-threaded environment).
- Slower than **HashMap** and **LinkedHashMap** due to sorting overhead.

TreeMap Syntax

```
TreeMap<KeyType, ValueType> mapName = new TreeMap<>();
```

Commonly Used Methods

Method	Description
<code>put(K key, V value)</code>	Adds a key-value pair.
<code>get(Object key)</code>	Returns value associated with the key.
<code>remove(Object key)</code>	Removes the entry by key.
<code>containsKey(Object key)</code>	Checks if a key exists.
<code>containsValue(Object value)</code>	Checks if a value exists.
<code>size()</code>	Returns number of key-value pairs.
<code>clear()</code>	Removes all entries.
<code>firstKey()</code>	Returns the first (lowest) key.
<code>lastKey()</code>	Returns the last (highest) key.
<code>headMap(K toKey)</code>	Returns view of map whose keys are less than <code>toKey</code> .
<code>tailMap(K fromKey)</code>	Returns view of map whose keys are greater than or equal to <code>fromKey</code> .
<code>subMap(K fromKey, K toKey)</code>	Returns view of map whose keys are in the range.

Example: Basic TreeMap

```
import java.util.*;

public class TreeMapExample {
    public static void main(String[] args) {
        TreeMap<Integer, String> map = new TreeMap<>();
        map.put(3, "Banana");
        map.put(1, "Apple");
        map.put(2, "Mango");
```



```
        System.out.println("TreeMap: " + map);
    }
```

Output:

```
TreeMap: {1=Apple, 2=Mango, 3=Banana}
```

Explanation

The keys are automatically sorted in ascending order.

Example: Using firstKey(), lastKey(), headMap(), tailMap(), subMap()

```
import java.util.*;

public class TreeMapMethods {
    public static void main(String[] args) {
        TreeMap<Integer, String> map = new TreeMap<>();
        map.put(100, "Hundred");
        map.put(50, "Fifty");
        map.put(150, "One Fifty");
        map.put(75, "Seventy Five");

        System.out.println("Original Map: " + map);
        System.out.println("First Key: " + map.firstKey());
        System.out.println("Last Key: " + map.lastKey());
        System.out.println("Head Map (toKey=100): " + map.headMap(100));
        System.out.println("Tail Map (fromKey=100): " + map.tailMap(100));
        System.out.println("Sub Map (50 to 150): " + map.subMap(50, 150));
    }
}
```

Output:

Original Map: {50=Fifty, 75=Seventy Five, 100=Hundred, 150=One Fifty}



First Key: 50

Last Key: 150

Head Map (toKey=100): {50=Fifty, 75=Seventy Five}

Tail Map (fromKey=100): {100=Hundred, 150=One Fifty}

Sub Map (50 to 150): {50=Fifty, 75=Seventy Five, 100=Hundred}

Performance

Operation	Time Complexity
-----------	-----------------

put()	O(log n)
-------	----------

get()	O(log n)
-------	----------

remove()	O(log n)
----------	----------

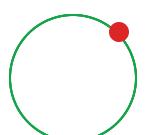
containsKey()	O(log n)
---------------	----------

TreeMap operations are slower than `HashMap` and `LinkedHashMap` because they require maintaining sorting order via a Red-Black Tree.

When to Use TreeMap

- When sorted ordering of keys is required.
- When you need range-based queries like `subMap()`, `headMap()`, etc.
- When you don't care about insertion order.

Conclusion





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Comparable Interface in Java

When we work with collections in Java, we often need to sort objects of our custom classes. But how does Java know which object is greater or lesser? This is where the **Comparable** interface comes to the rescue!

In this blog, we will explore what the **Comparable** interface is, why it is useful, and how to implement it with examples.

1. What is the Comparable Interface?

The **Comparable** interface in Java is used to define the natural ordering of objects. It allows objects of a class to be compared with each other based on a single property.

Syntax:

```
public interface Comparable<T> {  
    int compareTo(T o);  
}
```

Any class that implements **Comparable<T>** must provide an implementation for the **compareTo** (**T o**) method.

- If **this** object is **less than** the passed object → return a **negative value**



- If **this** object is **equal to** the passed object → return **zero**
- If **this** object is **greater than** the passed object → return a **positive value**

2. Why Use Comparable?

Problem Without Comparable:

Let's say we have a **Student** class and a list of students. If we try to sort them using **Collections.sort()**, Java won't know how to compare them.

```
import java.util.*;

class Student {
    int id;
    String name;

    Student(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public String toString() {
        return id + " - " + name;
    }
}

public class Main {
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
        students.add(new Student(3, "Alice"));
        students.add(new Student(1, "Bob"));
        students.add(new Student(2, "Charlie"));

        Collections.sort(students); // This will throw an error!
    }
}
```



Error:

```
Main.java:22: error: no suitable method found for sort(List<Student>)
Collections.sort(students);
```



Since Java doesn't know how to compare **Student** objects, it throws an error.

3. Implementing Comparable

To fix this, we implement **Comparable<Student>** and override **compareTo()**.

Sorting Students by ID:

```
import java.util.*;

class Student implements Comparable<Student> {
    int id;
    String name;

    Student(int id, String name) {
        this.id = id;
        this.name = name;
    }

    @Override
    public int compareTo(Student other) {
        return this.id - other.id; // Sorting in ascending order
    }

    @Override
    public String toString() {
        return id + " - " + name;
    }
}

public class Main {
```



```
public static void main(String[] args) {  
    List<Student> students = new ArrayList<>();  
    students.add(new Student(3, "Alice"));  
    students.add(new Student(1, "Bob"));  
    students.add(new Student(2, "Charlie"));  
  
    Collections.sort(students);  
  
    System.out.println(students);  
}
```

Output:

```
1 - Bob  
2 - Charlie  
3 - Alice
```

Now, the students are sorted by their **id** in ascending order.

4. Sorting in Descending Order

If you want to sort in **descending** order, just reverse the comparison:

```
@Override  
public int compareTo(Student other) {  
    return other.id - this.id; // Descending order  
}
```

Output:

```
3 - Alice  
2 - Charlie  
1 - Bob
```

5. Sorting by Name

What if we want to sort students alphabetically by name instead of ID?

```
@Override  
public int compareTo(Student other) {  
    return this.name.compareTo(other.name); // Sort by name  
}
```

Output:

```
Alice  
Bob  
Charlie
```

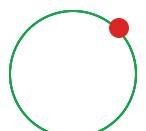
6. Key Points to Remember

- The **Comparable** interface is used to define **natural ordering**.
- Implement **compareTo()** in the class and define the sorting logic.
- **Collections.sort()** automatically uses **compareTo()** to sort objects.
- Return **negative**, **zero**, or **positive** values based on the comparison.

7. Alternative: Using Comparator

The **Comparable** interface is great when you need a **default sorting order**. But what if you want **multiple sorting criteria** (e.g., sort by ID or Name based on user choice)?

This is where **Comparator** comes in handy, which we'll discuss in next blog!





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Comparator Interface in Java

Sorting objects in Java is an essential skill, especially when dealing with collections of custom objects. In the previous blog, we learned about the **Comparable** interface, which defines the natural ordering of objects. But what if we want **multiple sorting criteria** (e.g., sorting by name, then by age)? This is where the **Comparator** interface comes in!

Let's explore what the **Comparator** interface is, why we need it, and how to implement it with examples.

1. What is the Comparator Interface?

The **Comparator** interface in Java allows us to define custom sorting logic for objects. Unlike **Comparable**, which enforces a natural ordering, **Comparator** gives us the flexibility to define multiple ways to compare objects.

Syntax:

```
import java.util.Comparator;  
  
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```



- If o1 should come before o2, return a **negative** number.
- If o1 is equal to o2, return zero.
- If o1 should come after o2, return a **positive** number.

2. Why Use Comparator?

Problem Without Comparator:

Let's say we have a **Student** class and we want to sort students by **name** and **age**. If we use **Comparable**, we can only define one sorting rule. But with **Comparator**, we can have multiple sorting criteria.

3. Sorting Using Comparator

Example 1: Sorting Students by Age

```
import java.util.*;  
  
class Student {  
    int id;  
    String name;  
    int age;  
  
    Student(int id, String name, int age) {  
        this.id = id;  
        this.name = name;  
        this.age = age;  
    }  
  
    @Override  
    public String toString() {  
        return id + " - " + name + " - Age: " + age;  
    }  
}
```

```

class AgeComparator implements Comparator<Student> {
    @Override
    public int compare(Student s1, Student s2) {
        return s1.age - s2.age; // Sorting by age in ascending order
    }
}

public class Main {
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
        students.add(new Student(3, "Alice", 22));
        students.add(new Student(1, "Bob", 20));
        students.add(new Student(2, "Charlie", 21));

        Collections.sort(students, new AgeComparator());

        System.out.println(students);
    }
}

```

Output:

```

1 - Bob - Age: 20
2 - Charlie - Age: 21
3 - Alice - Age: 22

```

4. Sorting by Name (Alphabetically)

```

class NameComparator implements Comparator<Student> {
    @Override
    public int compare(Student s1, Student s2) {
        return s1.name.compareTo(s2.name);
    }
}

```

Now, we can sort students by name:

```
Collections.sort(students, new NameComparator());
```



Output:

```
3 - Alice - Age: 22  
1 - Bob - Age: 20  
2 - Charlie - Age: 21
```



5. Sorting in Descending Order

To sort by age in descending order, simply reverse the subtraction:

```
@Override  
public int compare(Student s1, Student s2) {  
    return s2.age - s1.age; // Sorting by age in descending order  
}
```



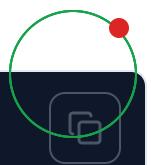
Output:

```
3 - Alice - Age: 22  
2 - Charlie - Age: 21  
1 - Bob - Age: 20
```



6. Using Lambda Expressions for Comparator (Java 8+)

Java 8 introduced **lambda expressions**, making sorting even easier:



```
Collections.sort(students, (s1, s2) -> s1.age - s2.age);
```

For descending order:

```
Collections.sort(students, (s1, s2) -> s2.age - s1.age);
```



Or even:

```
students.sort(Comparator.comparingInt(s -> s.age));
```



For sorting by name:

```
students.sort(Comparator.comparing(s -> s.name));
```



7. Sorting by Multiple Criteria

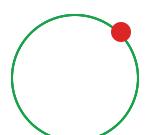
What if we want to sort by **name first**, and then **age**?

```
students.sort(  
    Comparator.comparing(Student::getName)  
        .thenComparingInt(Student::getAge)  
);
```



Explanation:

1. Sorts by **name** first.
2. If names are the same, sorts by **age**.



Java Tutorial – A Comprehensive Guide for Beginners

Introduction

Java is one of the most widely used and versatile programming languages in the world. From desktop applications to web services and mobile apps to enterprise systems, Java powers countless technologies and platforms.

This Java tutorial is designed for beginners who want to build a strong foundation in Java programming. Whether you're completely new to programming or transitioning from another language, this guide covers everything from core concepts to essential object-oriented principles and beyond.

If you're aiming to become a proficient Java developer, follow this structured path to master Java, step by step.

Prerequisites

Before you dive into Java, it's helpful (but not mandatory) to be familiar with:

- **Basic Computer Knowledge:** Understanding how software runs, what an IDE is, etc.
- **Logic Building:** Basic problem-solving and logical thinking.



- **Understanding of Programming Concepts** (optional): Knowing variables, loops, or functions from another language is a bonus.

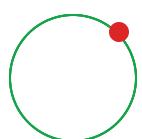
Master These Topics

Once you're ready, follow this structured list to learn Java from the ground up:

1. Introduction to Java
2. Java Fundamentals
3. Java Flow Control
4. Java Arrays
5. Java Methods
6. Java Object-Oriented Programming
7. Java Inner Classes
8. Static and Final Keywords
9. Java Exception Handling
10. Java Generics
11. Java Lambda Expressions
12. Java Collection Framework
13. Java Input/Output Streams
14. Java Reader/Writer

Introduction to Java

1. [Introduction to Java](#)
2. [Installing JDK on your computer](#)



[3. Java Comments](#)

Java Fundamentals

- [1. Java Variables and Literals](#)
- [2. Data Types in Java](#)
- [3. Operators in Java](#)
- [4. Java Basic Input and Output](#)
- [5. Java Expressions, Statements and Blocks](#)

Java Flow Control

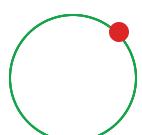
- [1. Java if else statement](#)
- [2. Java Ternary Operator](#)
- [3. Java For Loop](#)
- [4. Java while and do while loop](#)
- [5. Java continue and break statement](#)
- [6. Java Switch statement](#)

Java Arrays

- [1. Java Arrays](#)
- [2. Java Multidimensional Arrays](#)
- [3. Java Copy Arrays](#)

Java Methods

- [1. Methods in Java](#)



2. Parameter Passing in Java
3. Method Overloading
4. Variable Arguments
5. Java CommandLine Arguments
6. Java Recursive Method

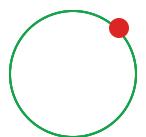
Java Object-Oriented Programming

1. Principles of Object-Oriented Programming
2. Java Class and Objects
3. Java Constructor
4. Encapsulation and Data Hiding
5. Inheritance in Java
6. Constructors in Inheritance
7. this Vs super Keyword
8. Method Overriding
9. Dynamic Method Dispatch
10. Polymorphism using Overloading and Overriding
11. Abstract Classes
12. Interfaces in Java

Java Inner Classes

1. Inner Classes in Java

Static and Final Keywords



1. Static Members and Static Blocks in Java
2. Final Members
3. Singleton Class

Java Exception Handling

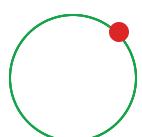
1. What are Exceptions
2. How to handle Exception
3. Try and Catch Block
4. Multiple and Nested Try Catch
5. Class Exception
6. Checked and Unchecked Exceptions
7. Throw Vs Throws
8. Finally Block
9. Try with Resources

Java Generics

1. Introduction to Generics
2. Defining Generic Class
3. Bounds on Generics
4. Java Generic Methods

Java Lambda Expressions

1. Introduction to Lambda Expressions
2. Parameters in Lambda Expressions



3. Capture in Lambda Expression

4. Method References

Java Collection Framework

1. Collections in Java

2. Understanding Collection Interface

3. Understanding List Interface in Java

4. Java ArrayList

5. Java LinkedList

6. Java Vector

7. Java Stack

8. Understanding Queue Interface in Java

9. Java PriorityQueue

10. Java ArrayDeque

11. Understanding Set Interface in Java

12. Java HashSet

13. Java LinkedHashSet

14. Java TreeSet

15. Understanding Map Interface in Java

16. Java TreeMap



Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Java Input/Output Streams

1. [What are Streams](#)
2. [InputStream and OutputStream Classes](#)
3. [FileInputStream and FileOutputStream](#)
4. [ByteArrayInputStream and ByteArrayOutputStream](#)
5. [ObjectInputStream and ObjectOutputStream](#)
6. [BufferedInputStream and BufferedOutputStream](#)
7. [Java PrintStream Class](#)

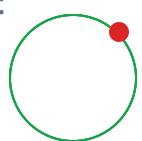
Java Reader/Writer

1. [Java File Class](#)
2. [Java Reader and Writer Class](#)
3. [InputStreamReader and OutputStreamWriter Class](#)
4. [FileReader and FileWriter Class](#)
5. [BufferedReader and BufferedWriter Class](#)
6. [StringReader and StringWriter Class](#)
7. [Java PrintWriter Class](#)

Key Features of Java

Java has remained popular for decades for good reason. Here's what makes it special:

- **Platform Independent:** Write once, run anywhere (thanks to the JVM).





 Pay Day Sale is Now Live!  Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



 Java Programming Handbook



What are Streams in Java?

Handling input and output (I/O) is an essential part of Java programming. Whether you're reading user input, writing data to files, or processing network connections, Java provides a powerful I/O system using **Streams**.

In this blog, we'll explore:

- What Streams are
- Why we need them
- How they work
- Types of Streams in Java
- Simple examples to understand their usage

1. What are Streams?

A **Stream** in Java is a sequence of data that flows from a **source** to a **destination**. Think of it like a pipeline that carries bytes of data from one place to another. Streams abstract the complexity of data transfer, making it easier to handle input and output operations.

Example Analogy:

- A **water pipe** carries water from a tank (source) to a tap (destination).



- A **Java Stream** carries data from an input source (keyboard, file, network) to an output destination (console, file, network).

2. Why Do We Need Streams?

Before Streams, handling I/O operations required dealing with low-level details like buffering, character encoding, and manual data transfers. Streams simplify this by:

- Providing a **consistent** way to handle different types of input and output.
- Automatically managing **buffering** and **data conversion**.
- Enabling **efficient** data transfer with minimal effort.

Without Streams (Using Arrays)

```
char[] data = {'H', 'e', 'l', 'l', 'o'};
for (char c : data) {
    System.out.print(c);
}
```



- This approach works but lacks flexibility for reading files or network data.

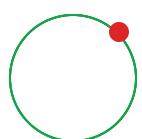
With Streams (Using System.out)

```
System.out.print("Hello");
```



- Java internally uses a **PrintStream** to send the output to the console.
- This is a **stream-based approach**, making it scalable and efficient.

3. Types of Streams in Java



Java I/O Streams are categorized into two broad types:

A. Byte Streams (for handling binary data)

- Deals with raw binary data (images, audio, videos, etc.).
- Uses **InputStream** and **OutputStream** classes.
- Reads/Writes **one byte at a time**.
- Example: **FileInputStream**, **FileOutputStream**.

B. Character Streams (for handling text data)

- Designed for reading and writing character data.
- Uses **Reader** and **Writer** classes.
- Reads/Writes **one character at a time**.
- Example: **FileReader**, **FileWriter**.

Stream Type	Class Hierarchy	Purpose
Byte Streams	InputStream , OutputStream	Handles binary data (images, audio, video)
Character Streams	Reader , Writer	Handles text-based data (UTF-8, ASCII)

4. Simple Example: Reading Input with Streams

Let's take a basic example using **System.in**, which is an **InputStream** for reading user input.

```
import java.io.IOException;

public class InputExample {
    public static void main(String[] args) throws IOException {
        System.out.println("Enter a character: ");
```

```
int input = System.in.read(); // Reads a single byte
System.out.println("You entered: " + (char) input);
}
```

Output:

```
Enter a character:
```



```
A
```

```
You entered: A
```

Explanation:

- `System.in.read()` reads one byte from the keyboard.
- Since it reads an integer value (ASCII code), we **cast it** to a character.

5. Understanding Data Flow in Streams

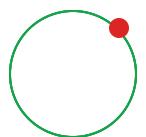
A **stream pipeline** in Java follows a simple flow:

Source → Stream → Destination

- **Input Stream:** Reads data from a source (file, keyboard, network).
- **Processing:** Optional transformations, like buffering or filtering.
- **Output Stream:** Writes data to a destination (console, file, network).

Example:

1. Keyboard (`System.in`) → InputStream → Program → OutputStream → Console (`System.out`)



2. File (FileInputStream) → InputStream → Program → OutputStream → Network (Socket OutputStream)

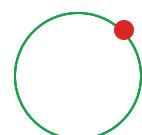
6. Key Differences: InputStream vs. OutputStream

Feature	InputStream	OutputStream
Purpose	To read data (input)	To write data (output)
Direction	Reads data from a source	Writes data to a destination
Inheritance Base	<code>java.io.InputStream</code> (abstract class)	<code>java.io.OutputStream</code> (abstract class)
Data Type Handled	Reads binary data (bytes)	Writes binary data (bytes)
Common Subclasses	<code>FileInputStream</code> , <code>ByteArrayInputStream</code> , <code>BufferedInputStream</code>	<code>FileOutputStream</code> , <code>ByteArrayOutputStream</code> , <code>BufferedOutputStream</code>

7. When to Use Streams in Java?

- Use Byte Streams when dealing with **binary files** like images, videos, or serialized objects.
- Use Character Streams when working with **text-based data** (CSV, JSON, XML, etc.).
- Use Buffered Streams when **performance matters** (reduces I/O overhead by buffering data).
- Use Object Streams when working with **serialization and deserialization** of Java objects.

Conclusion





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Java InputStream and OutputStream Classes

When working with files, network connections, or any other data sources in Java, you need a way to read and write data efficiently. This is where Java's **InputStream** and **OutputStream** classes come into play. They form the foundation of Java's I/O (Input/Output) system for handling byte-based data.

In this blog, we will explore these two fundamental classes, understand their purpose, and list the key classes that extend them. This will set the stage for our detailed exploration of specific stream classes in upcoming blogs.

1. Understanding InputStream and OutputStream

Java follows a stream-based approach to handling input and output. A **stream** is a sequence of data that flows from a source to a destination.

- **InputStream** is used for reading data (input) from a source.
- **OutputStream** is used for writing data (output) to a destination.

Both of these are abstract classes present in the **java.io** package, meaning they provide a blueprint for specific stream classes to extend and implement functionality.

2. Methods in InputStream Class



The **InputStream** class provides methods to read bytes from a source. Here are the key methods it defines:

- **int read()**

- Reads a single byte of data and returns it as an **int**. Returns **1** if the end of the stream is reached.

- **int read(byte[] b)**

- Reads bytes into the provided byte array **b** and returns the number of bytes read.

- **int read(byte[] b, int off, int len)**

- Reads up to **len** bytes into the array starting at **off** index.

- **void close()**

- Closes the stream and releases resources.

- **int available()**

- Returns the number of available bytes that can be read without blocking.

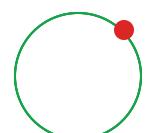
- **long skip(long n)**

- Skips **n** bytes and returns the actual number of bytes skipped.

3. Methods in OutputStream Class

The **OutputStream** class provides methods to write bytes to a destination. Here are the key methods it defines:

- **void write(int b)**



- Writes a single byte to the output stream.

- **void write(byte[] b)**

- Writes an entire byte array to the output stream.

- **void write(byte[] b, int off, int len)**

- Writes **len** bytes from the array starting at index **off**.

- **void flush()**

- Forces any buffered output to be written immediately.

- **void close()**

- Closes the stream and releases any resources.

4. Classes Extending InputStream

Several classes extend **InputStream** to provide specific functionalities. These will be covered in detail in upcoming blogs:

- **FileInputStream** - Reads data from a file.
- **ByteArrayInputStream** - Reads data from a byte array.
- **ObjectInputStream** - Reads objects from a stream (used in serialization).
- **BufferedInputStream** - Wraps another **InputStream** to improve performance.

5. Classes Extending OutputStream

Similar to **InputStream**, the **OutputStream** class has multiple implementations for different use cases:



Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



FileInputStream and FileOutputStream

In the previous blog, we covered the basics of **InputStream** and **OutputStream** classes. Now, let's dive into two essential classes that extend them: **FileInputStream** and **FileOutputStream**. These classes allow us to read from and write to files in Java.

1. Understanding FileInputStream and FileOutputStream

- **FileInputStream** is used to read raw byte data from a file.
- **FileOutputStream** is used to write raw byte data to a file.

Since these classes deal with bytes, they are suitable for handling binary files such as images, audio, and videos. However, they can also process text files, though character streams (like **FileReader** and **FileWriter**) are generally preferred for that purpose.

2. FileInputStream Class

The **FileInputStream** class allows reading data from a file as a stream of bytes. It extends **InputStream** and provides methods to read bytes from a file.

• Creating a FileInputStream

To create an instance of **FileInputStream**, you can pass the file path as a string or a **File** object:



```
import java.io.*;

public class FileInputStreamExample {
    public static void main(String[] args) {
        try (FileInputStream fis = new FileInputStream("example.txt")) {
            int data;
            while ((data = fis.read()) != -1) {
                System.out.print((char) data); // Reading byte by byte
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```



● Important Methods in FileInputStream

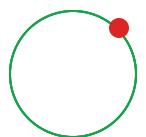
- `int read()` - Reads one byte at a time and returns it.
- `int read(byte[] b)` - Reads bytes into the given byte array.
- `int read(byte[] b, int off, int len)` - Reads up to `len` bytes into an array starting at index `off`.
- `void close()` - Closes the stream and releases file resources.
- `int available()` - Returns the number of bytes available for reading.

3. FileOutputStream Class

The `FileOutputStream` class allows writing byte data to a file. It extends `OutputStream` and provides methods to write data into a file.

● Creating a FileOutputStream

To write data into a file, we create an instance of `FileOutputStream`:



```
import java.io.*;

public class FileOutputStreamExample {
    public static void main(String[] args) {
        String data = "Hello, Java I/O!";
        try (FileOutputStream fos = new FileOutputStream("output.txt")) {
            fos.write(data.getBytes());
            System.out.println("Data written successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

• Important Methods in FileOutputStream

- **void write(int b)** - Writes a single byte.
- **void write(byte[] b)** - Writes an array of bytes.
- **void write(byte[] b, int off, int len)** - Writes **len** bytes starting at index **off**.
- **void close()** - Closes the stream.
- **void flush()** - Forces any buffered output to be written immediately.

4. When to Use FileInputStream and FileOutputStream?

- Use **FileInputStream** when you need to read binary data (such as images, PDFs, etc.) or when working with raw bytes.
- Use **FileOutputStream** when you need to write raw bytes to a file, like saving an image or a binary file.
- If working with text files, consider using **FileReader** and **FileWriter** instead, as they handle character encoding properly.



Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Java I/O Streams: **ByteArrayInputStream** and **ByteArrayOutputStream**

In this blog, we will explore **ByteArrayInputStream** and **ByteArrayOutputStream**, two memory-based I/O stream classes in Java. These classes are useful when working with byte arrays instead of files or external data sources.

1. What Are **ByteArrayInputStream** and **ByteArrayOutputStream**?

- **ByteArrayInputStream** allows an application to read data from a byte array as an input stream.
- **ByteArrayOutputStream** allows writing data to a byte array, which grows automatically.

These streams are often used in testing, conversions, or working with binary data in memory without accessing the filesystem.

2. **ByteArrayInputStream** Methods (With Examples)

- **int read()**

Reads the next byte of data from the input stream.

```
import java.io.*;
```



```
public class ByteArrayInputStreamExample {  
    public static void main(String[] args) {  
        byte[] data = {65, 66, 67}; // A, B, C  
        try (ByteArrayInputStream bais = new ByteArrayInputStream(data)) {  
            int byteData;  
            while ((byteData = bais.read()) != -1) {  
                System.out.print((char) byteData);  
            }  
        }  
    }  
}
```

Output:

ABC

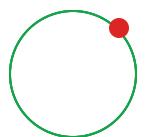


• **int read(byte[] b, int off, int len)**

Reads up to **len** bytes into **b** starting from offset **off**.

```
import java.io.*;  
  
public class ByteArrayInputStreamReadArray {  
    public static void main(String[] args) {  
        byte[] input = "Hello World".getBytes();  
        byte[] buffer = new byte[5];  
  
        try (ByteArrayInputStream bais = new ByteArrayInputStream(input)) {  
            int bytesRead = bais.read(buffer, 0, buffer.length);  
            System.out.println(new String(buffer, 0, bytesRead));  
        }  
    }  
}
```

Output:





- **int available()**

Returns the number of remaining bytes that can be read.

```
import java.io.*;  
  
public class ByteArrayInputStreamAvailable {  
    public static void main(String[] args) {  
        byte[] data = "Test".getBytes();  
  
        try (ByteArrayInputStream bais = new ByteArrayInputStream(data)) {  
            System.out.println("Bytes available: " + bais.available());  
        }  
    }  
}
```

Output:

```
Bytes available: 4
```



- **void reset()**

Resets the stream to the beginning.

```
import java.io.*;  
  
public class ByteArrayInputStreamReset {  
    public static void main(String[] args) {  
        byte[] data = "ABCD".getBytes();  
  
        try (ByteArrayInputStream bais = new ByteArrayInputStream(data)) {  
            System.out.print((char) bais.read()); // A  
        }  
    }  
}
```



```
        System.out.print((char) bais.read()); // B
        bais.reset();
        System.out.print((char) bais.read()); // A again
    }
}
```

Output:

ABA



3. ByteArrayOutputStream Methods (With Examples)

- **void write(int b)**

Writes a single byte to the output stream.

```
import java.io.*;

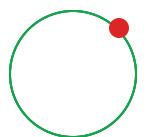
public class ByteArrayOutputStreamWriteSingle {
    public static void main(String[] args) {
        try (ByteArrayOutputStream baos = new ByteArrayOutputStream()) {
            baos.write(65); // 'A'
            System.out.println(baos.toString());
        }
    }
}
```

Output:

A



- **void write(byte[] b)**



Writes a byte array to the stream.

```
import java.io.*;  
  
public class ByteArrayOutputStreamWriteArray {  
    public static void main(String[] args) {  
        try (ByteArrayOutputStream baos = new ByteArrayOutputStream()) {  
            String msg = "Hello Java";  
            baos.write(msg.getBytes());  
            System.out.println(baos.toString());  
        }  
    }  
}
```

Output:

```
Hello Java
```

● **byte[] toByteArray()**

Returns the current contents as a byte array.

```
import java.io.*;  
  
public class ByteArrayOutputStreamToByteArray {  
    public static void main(String[] args) {  
        try (ByteArrayOutputStream baos = new ByteArrayOutputStream()) {  
            baos.write("Data".getBytes());  
            byte[] result = baos.toByteArray();  
            for (byte b : result) {  
                System.out.print((char) b);  
            }  
        }  
    }  
}
```

Output:

Data



- **void reset()**

Clears the current buffer.

```
import java.io.*;  
  
public class ByteArrayOutputStreamReset {  
    public static void main(String[] args) {  
        try (ByteArrayOutputStream baos = new ByteArrayOutputStream()) {  
            baos.write("Before Reset".getBytes());  
            baos.reset();  
            baos.write("After Reset".getBytes());  
            System.out.println(baos.toString());  
        }  
    }  
}
```

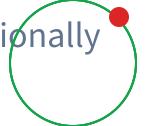
Output:

After Reset



4. When to Use ByteArrayInputStream and ByteArrayOutputStream?

- Use **ByteArrayInputStream** when reading from an in-memory byte array instead of from an external file.
- Use **ByteArrayOutputStream** when you want to collect bytes in memory and optionally convert them to a byte array or string.





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



Java I/O Streams: ObjectInputStream and ObjectOutputStream

Java provides **ObjectInputStream** and **ObjectOutputStream** classes for serialization and deserialization of objects. These classes are part of the **java.io** package and are used when you want to save the state of an object or send it over a network.

1. What Are ObjectInputStream and ObjectOutputStream?

- **ObjectOutputStream**: Used to serialize Java objects into a stream of bytes and write them to an OutputStream (like a file).
- **ObjectInputStream**: Used to deserialize the stream of bytes into a Java object from an InputStream.

These streams are commonly used when storing objects in files or transmitting them over the network.

2. ObjectOutputStream Methods (With Examples)

• **void writeObject(Object obj)**

Serializes the given object and writes it to the output stream.



Example:

```
import java.io.*;  
  
class Student implements Serializable {  
    int id;  
    String name;  
  
    public Student(int id, String name) {  
        this.id = id;  
        this.name = name;  
    }  
}  
  
public class ObjectOutputExample {  
    public static void main(String[] args) {  
        Student student = new Student(1, "John");  
  
        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("student.ser"))){  
            oos.writeObject(student);  
            System.out.println("Object has been serialized.");  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Output:

```
Object has been serialized.
```

(File **student.ser** contains serialized byte data of the **Student** object)

Something like this :

? ?? ? sr ? Student ? ?? ? I ? ? / G ? I ? idL ? name t Ljava/lang/String; xp ? ?? ? t ? John

3. ObjectInputStream Methods (With Examples)

- **Object readObject()**

Reads an object from the input stream and deserializes it.

Example:

```
import java.io.*;  
  
class Student implements Serializable {  
    int id;  
    String name;  
  
    public Student(int id, String name) {  
        this.id = id;  
        this.name = name;  
    }  
}  
  
public class ObjectInputExample {  
    public static void main(String[] args) {  
        try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("student.ser"))){  
            Student student = (Student) ois.readObject();  
            System.out.println("ID: " + student.id);  
            System.out.println("Name: " + student.name);  
        } catch (IOException | ClassNotFoundException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Output:

```
ID: 1  
Name: John
```



Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



Java I/O Streams: `BufferedInputStream` and `BufferedOutputStream`

In this blog, we'll dive into two important Java classes that help improve performance during file I/O operations: `BufferedInputStream` and `BufferedOutputStream`. These classes add buffering capabilities to reduce the number of disk accesses, making I/O operations more efficient.

1. What Are `BufferedInputStream` and `BufferedOutputStream`?

- `BufferedInputStream`: Wraps an `InputStream` and adds a buffer to reduce read operations from the underlying source.
- `BufferedOutputStream`: Wraps an `OutputStream` and adds a buffer to reduce write operations to the underlying destination.

These are especially useful when working with large files or when performance is a concern.

2. `BufferedInputStream` Methods (With Examples)

- `int read()`

Reads a single byte of data. Returns `-1` at the end of the file.



Example:

```
import java.io.*;  
  
public class BufferedInputStreamReadExample {  
    public static void main(String[] args) {  
        try (BufferedInputStream bis = new BufferedInputStream(new FileInputStream("input.txt"))){  
            int data;  
            while ((data = bis.read()) != -1) {  
                // Print each character read from the file  
                System.out.print((char) data);  
            }  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Output (if input.txt contains "Buffered Stream"):

Buffered Stream

• `int read(byte[] b, int off, int len)`

Reads up to `len` bytes of data into an array `b`, starting at offset `off`.

Example:

```
import java.io.*;  
  
public class BufferedInputStreamReadArrayExample {  
    public static void main(String[] args) {  
        byte[] buffer = new byte[20];  
        try (BufferedInputStream bis = new BufferedInputStream(new FileInputStream("input.txt"))){  
            int bytesRead = bis.read(buffer, 0, buffer.length);  
            System.out.println("Read " + bytesRead + " bytes: " + new String(buffer));  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Output:

Read 16 bytes: Buffered Stream



• void close()

Closes the stream and releases any resources associated with it.

Example:

```
import java.io.*;

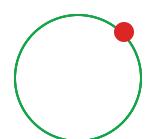
public class BufferedInputStreamCloseExample {
    public static void main(String[] args) {
        try (BufferedInputStream bis = new BufferedInputStream(new FileInputStream("file.txt"))) {
            System.out.println("Stream opened. It will be closed automatically");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Output:

Stream opened. It will be closed automatically after the try block.



3. BufferedOutputStream Methods (With Examples)



- **void write(int b)**

Writes a single byte to the output stream.

Example:

```
import java.io.*;  
  
public class BufferedOutputStreamWriteExample {  
    public static void main(String[] args) {  
        try (BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream("output.txt"))){  
            bos.write(66); // ASCII for 'B'  
            bos.flush(); // Ensures data is written to the file immediately  
            System.out.println("Single byte written successfully.");  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Output:

```
Single byte written successfully.
```

(File **output.txt** contains: **B**)

- **void write(byte[] b, int off, int len)**

Writes **len** bytes from the array **b**, starting at offset **off**.

Example:

```
import java.io.*;  
  
public class BufferedOutputStreamWriteArrayExample {
```

```
public static void main(String[] args) {  
    String data = "Buffered Output Example";  
    byte[] bytes = data.getBytes();  
  
    try (BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream("output.txt"))){  
        bos.write(bytes, 0, bytes.length); // Write all bytes  
        bos.flush(); // Push any remaining buffered data to file  
        System.out.println("Byte array written successfully.");  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```

Output:

Byte array written successfully.



(File **output.txt** contains: **Buffered Output Example**)

• **void flush()**

Flushes the buffered output to the file. Ensures no data is stuck in memory.

“Note: flush() is especially useful when writing to a stream that stays open, like network sockets or large files.”

• **void close()**

Closes the stream and flushes any remaining data.

Example:

```
import java.io.*;  
  
public class BufferedOutputStreamCloseExample {
```





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Java PrintStream Class

The **PrintStream** class in Java is a specialized output stream used to print data in a human-readable format. It extends the **FilterOutputStream** class and adds functionality to write formatted representations of objects to the output stream.

1. Why Use PrintStream?

While **FileOutputStream** and **BufferedOutputStream** work well for raw byte operations, **PrintStream** is ideal for:

- Printing textual representations of data
- Automatically flushing output
- Writing formatted output without needing manual conversions

You might already be familiar with it — **System.out** is a **PrintStream** object!

2. Creating a PrintStream Object

You can create a **PrintStream** using various constructors:

```
PrintStream ps = new PrintStream(String fileName);
PrintStream ps = new PrintStream(File file);
```



3. Key Methods in PrintStream (With Examples)

- **void print(String s)**

Prints a string to the stream.

```
import java.io.*;  
  
public class PrintStreamPrintExample {  
    public static void main(String[] args) throws FileNotFoundException {  
        // Creating a PrintStream linked to output.txt  
        PrintStream ps = new PrintStream("output.txt");  
  
        // Writing plain text to the file  
        ps.print("Hello from PrintStream!");  
  
        ps.close();  
    }  
}
```

Output (content inside output.txt):

```
Hello from PrintStream!
```

- **void println(String s)**

Prints a string followed by a newline character.

```
import java.io.*;  
  
public class PrintStreamPrintlnExample {  
    public static void main(String[] args) throws FileNotFoundException {  
        PrintStream ps = new PrintStream("output.txt");  
    }  
}
```

```
// Printing lines with newLine  
ps.println("Line 1");  
ps.println("Line 2");  
  
ps.close();  
}  
}
```

Output:

```
Line 1  
Line 2
```



• **void printf(String format, Object... args)**

Prints a formatted string using the specified format and arguments.

```
import java.io.*;  
  
public class PrintStreamPrintfExample {  
    public static void main(String[] args) throws FileNotFoundException {  
        PrintStream ps = new PrintStream("output.txt");  
  
        // Writing formatted text  
        ps.printf("Name: %s, Age: %d", "Alice", 25);  
  
        ps.close();  
    }  
}
```

Output:

```
Name: Alice, Age: 25
```



- **void write(int b)**

Writes the specified byte to the stream. Useful for writing binary data.

```
import java.io.*;  
  
public class PrintStreamWriteExample {  
    public static void main(String[] args) throws FileNotFoundException {  
        PrintStream ps = new PrintStream("output.txt");  
  
        // Writes character 'A' (ASCII value 65)  
        ps.write(65);  
  
        ps.close();  
    }  
}
```

Output:

```
A
```

- **void flush()**

Forces any buffered output bytes to be written out. Usually not needed when using `println`, but good to know.

```
import java.io.*;  
  
public class PrintStreamFlushExample {  
    public static void main(String[] args) throws FileNotFoundException {  
        PrintStream ps = new PrintStream("output.txt");  
  
        ps.print("Data not yet flushed");  
  
        // Manually flush to ensure data is written immediately  
        ps.flush();  
}
```



```
        ps.close();
    }
}
```

- **void close()**

Closes the stream and releases any system resources associated with it.

```
import java.io.*;

public class PrintStreamCloseExample {
    public static void main(String[] args) throws FileNotFoundException {
        PrintStream ps = new PrintStream("output.txt");

        ps.print("Closing the stream now.");
        ps.close(); // Recommended to close to release file resources
    }
}
```

4. Special Note: System.out is a PrintStream

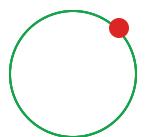
You use it every day without realizing:

```
System.out.println("This is a PrintStream!");
```

It behaves just like the examples shown above.

5. When Should You Use PrintStream?

- When you want to print nicely formatted, human-readable output
- When writing data to files and you want to use print/println/printf features



Java Tutorial – A Comprehensive Guide for Beginners

Introduction

Java is one of the most widely used and versatile programming languages in the world. From desktop applications to web services and mobile apps to enterprise systems, Java powers countless technologies and platforms.

This Java tutorial is designed for beginners who want to build a strong foundation in Java programming. Whether you're completely new to programming or transitioning from another language, this guide covers everything from core concepts to essential object-oriented principles and beyond.

If you're aiming to become a proficient Java developer, follow this structured path to master Java, step by step.

Prerequisites

Before you dive into Java, it's helpful (but not mandatory) to be familiar with:

- **Basic Computer Knowledge:** Understanding how software runs, what an IDE is, etc.
- **Logic Building:** Basic problem-solving and logical thinking.



- **Understanding of Programming Concepts** (optional): Knowing variables, loops, or functions from another language is a bonus.

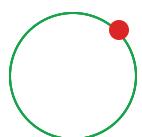
Master These Topics

Once you're ready, follow this structured list to learn Java from the ground up:

1. Introduction to Java
2. Java Fundamentals
3. Java Flow Control
4. Java Arrays
5. Java Methods
6. Java Object-Oriented Programming
7. Java Inner Classes
8. Static and Final Keywords
9. Java Exception Handling
10. Java Generics
11. Java Lambda Expressions
12. Java Collection Framework
13. Java Input/Output Streams
14. Java Reader/Writer

Introduction to Java

1. [Introduction to Java](#)
2. [Installing JDK on your computer](#)



[3. Java Comments](#)

Java Fundamentals

- [1. Java Variables and Literals](#)
- [2. Data Types in Java](#)
- [3. Operators in Java](#)
- [4. Java Basic Input and Output](#)
- [5. Java Expressions, Statements and Blocks](#)

Java Flow Control

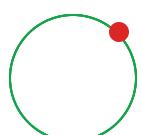
- [1. Java if else statement](#)
- [2. Java Ternary Operator](#)
- [3. Java For Loop](#)
- [4. Java while and do while loop](#)
- [5. Java continue and break statement](#)
- [6. Java Switch statement](#)

Java Arrays

- [1. Java Arrays](#)
- [2. Java Multidimensional Arrays](#)
- [3. Java Copy Arrays](#)

Java Methods

- [1. Methods in Java](#)



2. Parameter Passing in Java
3. Method Overloading
4. Variable Arguments
5. Java CommandLine Arguments
6. Java Recursive Method

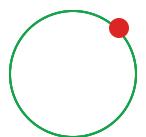
Java Object-Oriented Programming

1. Principles of Object-Oriented Programming
2. Java Class and Objects
3. Java Constructor
4. Encapsulation and Data Hiding
5. Inheritance in Java
6. Constructors in Inheritance
7. this Vs super Keyword
8. Method Overriding
9. Dynamic Method Dispatch
10. Polymorphism using Overloading and Overriding
11. Abstract Classes
12. Interfaces in Java

Java Inner Classes

1. Inner Classes in Java

Static and Final Keywords



1. Static Members and Static Blocks in Java
2. Final Members
3. Singleton Class

Java Exception Handling

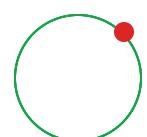
1. What are Exceptions
2. How to handle Exception
3. Try and Catch Block
4. Multiple and Nested Try Catch
5. Class Exception
6. Checked and Unchecked Exceptions
7. Throw Vs Throws
8. Finally Block
9. Try with Resources

Java Generics

1. Introduction to Generics
2. Defining Generic Class
3. Bounds on Generics
4. Java Generic Methods

Java Lambda Expressions

1. Introduction to Lambda Expressions
2. Parameters in Lambda Expressions



3. Capture in Lambda Expression

4. Method References

Java Collection Framework

1. Collections in Java

2. Understanding Collection Interface

3. Understanding List Interface in Java

4. Java ArrayList

5. Java LinkedList

6. Java Vector

7. Java Stack

8. Understanding Queue Interface in Java

9. Java PriorityQueue

10. Java ArrayDeque

11. Understanding Set Interface in Java

12. Java HashSet

13. Java LinkedHashSet

14. Java TreeSet

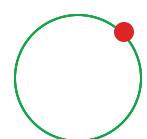
15. Understanding Map Interface in Java

16. Java HashMap

17. Java LinkedHashMap

18. Java TreeMap

19. Comparable Interface in Java



20. Comparator Interface in Java

Java Input/Output Streams

1. [What are Streams](#)
2. [InputStream and OutputStream Classes](#)
3. [FileInputStream and FileOutputStream](#)



Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#).

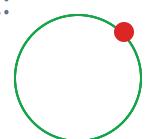
Java Reader/Writer

1. [Java File Class](#)
2. [Java Reader and Writer Class](#)
3. [InputStreamReader and OutputStreamWriter Class](#)
4. [FileReader and FileWriter Class](#)
5. [BufferedReader and BufferedWriter Class](#)
6. [StringReader and StringWriter Class](#)
7. [Java PrintWriter Class](#)

Key Features of Java

Java has remained popular for decades for good reason. Here's what makes it special:

- **Platform Independent:** Write once, run anywhere (thanks to the JVM).





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Java File Class in Detail

The `java.io.File` class is used to represent file and directory pathnames in an abstract manner. It doesn't represent the contents of a file but allows you to create, delete, check properties (like size, existence), and manipulate files or directories.

1. Constructors of File Class

```
File(String pathname)
File(String parent, String child)
File(File parent, String child)
```



Example:

```
import java.io.File;

public class FileConstructorExample {
    public static void main(String[] args) {
        // Creating File object using single path
        File file1 = new File("example.txt");

        // Creating File object using parent and child strings
        File file2 = new File("C:/Users", "example.txt");

        // Creating File object using parent File object and child
        File parentDir = new File("C:/Users");
```



```
File file3 = new File(parentDir, "example.txt");

System.out.println(file1.getPath());
System.out.println(file2.getPath());
System.out.println(file3.getPath());
}

}
```

Output:

```
example.txt
C:\\Users\\example.txt
C:\\Users\\example.txt
```

2. Commonly Used Methods with Examples

- **boolean createNewFile()**

Creates a new file if it doesn't exist.

```
import java.io.File;
import java.io.IOException;

public class FileCreateExample {
    public static void main(String[] args) {
        try {
            File file = new File("testfile.txt");
            if (file.createNewFile()) {
                System.out.println("File created: " + file.getName());
            } else {
                System.out.println("File already exists.");
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Output:

```
File created: testfile.txt
```



● boolean exists()

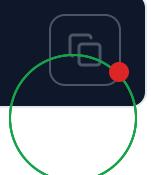
Checks whether the file exists.

```
File file = new File("testfile.txt");
System.out.println("File exists: " + file.exists());
```



Output:

```
File exists: true
```



- **boolean delete()**

Deletes the file or directory.

```
File file = new File("testfile.txt");
if (file.delete()) {
    System.out.println("File deleted successfully.");
} else {
    System.out.println("Failed to delete the file.");
}
```



Output:

```
File deleted successfully.
```



- **String getName(), String getPath(), String getAbsolutePath()**

These methods provide various representations of the file path.

```
File file = new File("testfile.txt");
System.out.println("Name: " + file.getName());
System.out.println("Path: " + file.getPath());
System.out.println("Absolute Path: " + file.getAbsolutePath());
```



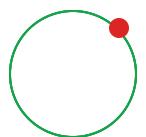
Output:

```
Name: testfile.txt
Path: testfile.txt
Absolute Path: C:\\your\\project\\directory\\testfile.txt
```



- **boolean isDirectory(), boolean isFile()**

Checks whether it is a directory or file.



```
File file = new File("testfile.txt");
System.out.println("Is file? " + file.isFile());
System.out.println("Is directory? " + file.isDirectory());
```



Output:

```
Is file? true
Is directory? false
```



● **String[] list()**

Returns the list of files and directories in a directory.

```
File dir = new File(".");
String[] files = dir.list();
if (files != null) {
    for (String name : files) {
        System.out.println(name);
    }
}
```



Output:

```
FileClassExample.java
anotherfile.txt
```



● **long length()**

Returns the file size in bytes.

```
File file = new File("testfile.txt");
System.out.println("File size: " + file.length() + " bytes");
```



Output:

```
File size: 25 bytes
```



• **boolean mkdir() and boolean mkdirs()**

Creates a single directory or multiple directories.

```
File dir1 = new File("newFolder");
File dir2 = new File("parentDir/childDir");
System.out.println("Single dir created: " + dir1.mkdir());
System.out.println("Multiple dirs created: " + dir2.mkdirs());
```



Output:

```
Single dir created: true
Multiple dirs created: true
```



3. When to Use the File Class?

- Use it when you need to interact with the file system—checking if files/directories exist, creating or deleting files, listing directory contents, etc.
- It acts as a foundation for many file handling operations and is frequently used with other I/O classes for reading or writing content.

Conclusion

In this blog, we explored the **File** class in Java, its constructors, and commonly used methods with practical examples and outputs. We saw how it helps us interact with the file system for operations like creation, deletion, and fetching metadata of files or directories.





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



Java Reader and Writer Classes in Java

In Java, when working with text-based input and output (character streams), the core classes to use are **Reader** and **Writer**. These are abstract classes provided in the **java.io** package that form the foundation for all character-based input and output operations.

Class Hierarchy Diagram



```
java.lang.Object
  └── java.io.Reader
      ├── BufferedReader
      ├── CharArrayReader
      ├── FilterReader
      ├── InputStreamReader
          └── FileReader
      ├── PipedReader
      └── StringReader

  └── java.io.Writer
      ├── BufferedWriter
      ├── CharArrayWriter
      ├── FilterWriter
      ├── OutputStreamWriter
          └── FileWriter
      ├── PipedWriter
      ├── PrintWriter
      └── StringWriter
```

1. What Are Reader and Writer Classes?

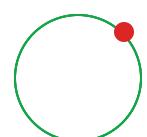
- **Reader**: Abstract class for reading character streams.
- **Writer**: Abstract class for writing character streams.

They are used when working with **text data**, unlike **InputStream** and **OutputStream** which work with **binary data**.

2. Reader Class Methods (With Examples)

int read()

Reads a single character. Returns -1 at the end of the stream.



```
import java.io.*;

public class ReaderReadExample {
    public static void main(String[] args) {
        try (Reader reader = new StringReader("Hello")) {
            int ch;
            while ((ch = reader.read()) != -1) {
                System.out.print((char) ch); // Prints each character
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```



Output:

```
Hello
```



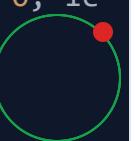
Explanation: **StringReader** is used here to simulate a stream. It reads characters one by one.

int read(char[] cbuf)

Reads characters into an array and returns the number of characters read.

```
import java.io.*;

public class ReaderReadArrayExample {
    public static void main(String[] args) {
        try (Reader reader = new StringReader("Java I/O")) {
            char[] buffer = new char[10];
            int len = reader.read(buffer);
            System.out.println("Read characters: " + new String(buffer, 0, le
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```



```
    }  
}
```

Output:

```
Read characters: Java I/O
```



void close()

Closes the reader and releases any associated system resources.

```
import java.io.*;  
  
public class ReaderCloseExample {  
    public static void main(String[] args) {  
        try (Reader reader = new StringReader("Close Example")) {  
            reader.read();  
            System.out.println("Reader closed automatically using try-with-re  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```



Output:

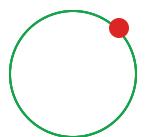
```
Reader closed automatically using try-with-resources
```



3. Writer Class Methods (With Examples)

void write(int c)

Writes a single character.



```
import java.io.*;

public class WriterWriteCharExample {
    public static void main(String[] args) {
        try (Writer writer = new FileWriter("output1.txt")) {
            writer.write(65); // Writes character 'A'
            System.out.println("Single character written.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Output:

```
Single character written.
```

(File **output1.txt** contains: A)

void write(char[] cbuf)

Writes an array of characters.



```
import java.io.*;

public class WriterWriteCharArrayExample {
    public static void main(String[] args) {
        char[] data = { 'J', 'a', 'v', 'a' };
        try (Writer writer = new FileWriter("output2.txt")) {
            writer.write(data);
            System.out.println("Character array written.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Output:

Character array written.



(File **output2.txt** contains: Java)

void write(String str)

Writes an entire string.



```
import java.io.*;

public class WriterWriteStringExample {
    public static void main(String[] args) {
        try (Writer writer = new FileWriter("output3.txt")) {
            writer.write("Java Writer Example");
            System.out.println("String written successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Output:

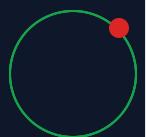
String written successfully.



(File **output3.txt** contains: Java Writer Example)

void flush()

Flushes the writer, forcing any buffered output to be written.



```
import java.io.*;

public class WriterFlushExample {
    public static void main(String[] args) {
        try (Writer writer = new FileWriter("output4.txt")) {
            writer.write("Flushed output.");
            writer.flush();
            System.out.println("Output flushed.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Output:

Output flushed.

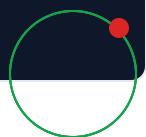


void close()

Closes the stream and releases system resources.

```
import java.io.*;

public class WriterCloseExample {
    public static void main(String[] args) {
        try (Writer writer = new FileWriter("output5.txt")) {
            writer.write("Closing stream");
            System.out.println("Writer closed using try-with-resources.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



InputStreamReader and OutputStreamWriter Classes

In Java, **InputStreamReader** and **OutputStreamWriter** are bridge classes that convert byte streams to character streams and vice versa. They are essential when you need to read or write text using specific character encodings.

1. Class Hierarchy

```
java.lang.Object
  ↗ java.io.Reader
    ↗ java.io.InputStreamReader

java.lang.Object
  ↗ java.io.Writer
    ↗ java.io.OutputStreamWriter
```



- **InputStreamReader** extends **Reader**
- **OutputStreamWriter** extends **Writer**

2. InputStreamReader Class

- Constructor



```
InputStreamReader(InputStream in)
InputStreamReader(InputStream in, Charset charset)
InputStreamReader(InputStream in, String charsetName)
```

- Example: Reading a file with InputStreamReader

```
import java.io.*;

public class InputStreamReaderExample {
    public static void main(String[] args) {
        try (InputStreamReader reader = new InputStreamReader(new FileInputStream("input.txt")));
            int data;
            while ((data = reader.read()) != -1) {
                // Convert byte to char and print
                System.out.print((char) data);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Output (If input.txt contains "Hello World"):

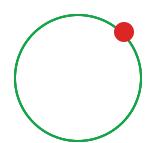
```
Hello World
```

Explanation:

- Reads bytes from file, decodes them into characters using the platform's default charset.
- **try-with-resources** auto-closes the stream.

- **int read()**

Reads a single character.



- **int read(char[] cbuf, int offset, int length)**

Reads characters into an array with offset and length.

```
import java.io.*;  
  
public class InputStreamReaderReadArray {  
    public static void main(String[] args) {  
        char[] buffer = new char[20];  
        try (InputStreamReader reader = new InputStreamReader(new FileInputStream("HelloWorld.txt"))){  
            int charsRead = reader.read(buffer, 0, buffer.length);  
            System.out.println("Read " + charsRead + " characters: " + new String(buffer, 0, charsRead));  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Output:

```
Read 11 characters: Hello World
```

- **void close()**

Closes the stream and releases any resources. If you're using **try-with-resources**, this is called automatically.

Example with explicit close (Not recommended):

```
import java.io.*;  
  
public class InputStreamReaderCloseExample {  
    public static void main(String[] args) {  
        InputStreamReader reader = null;  
        //Note:- Try-with-resources ensures the stream is closed automatically  
        try {  
            reader = new InputStreamReader(new FileInputStream("HelloWorld.txt"));  
            int charsRead = reader.read(buffer, 0, buffer.length);  
            System.out.println("Read " + charsRead + " characters: " + new String(buffer, 0, charsRead));  
        } catch (IOException e) {  
            e.printStackTrace();  
        } finally {  
            if (reader != null) {  
                reader.close();  
            }  
        }  
    }  
}
```

```
    reader = new InputStreamReader(new FileInputStream("input.txt"));
    System.out.println("Stream is open.");
    // You can perform operations here
    reader.close(); // Must be closed manually
    System.out.println("Stream is now closed.");
} catch (IOException e) {
    e.printStackTrace();
}
}
```

Output:

```
Stream is open.
Stream is now closed.
```



3. OutputStreamWriter Class

- Constructor

```
OutputStreamWriter(OutputStream out)
OutputStreamWriter(OutputStream out, Charset charset)
OutputStreamWriter(OutputStream out, String charsetName)
```



- void write(int c) – Writes a single character

```
import java.io.*;

public class OutputStreamWriterWriteChar {
    public static void main(String[] args) {
        try (OutputStreamWriter writer = new OutputStreamWriter(new FileOutput
            writer.write(65); // Writes character 'A' to the output.txt file
            System.out.println("Character written.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```



```
        }
    }
}
```

Output:

```
Character written.
```



File content (output.txt):

```
A
```

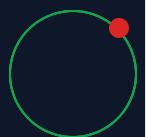


Explanation:

- The integer **65** is the ASCII value of character '**'A'**'.
 - This method writes the corresponding character to the file.
 - Since we used try-with-resources, the stream is automatically closed.
- **void write(char[] cbuf, int off, int len)** – Writes characters from a char array

```
import java.io.*;

public class OutputStreamWriterWriteArray {
    public static void main(String[] args) {
        char[] chars = "Hello Java".toCharArray();
        try (OutputStreamWriter writer = new OutputStreamWriter(new FileOutpu
writer.write(chars, 0, chars.length);
        System.out.println("Character array written.");
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```



Output:

```
Character array written.
```



File content (`output.txt`):

```
Hello Java
```

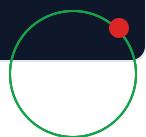


Explanation:

- Converts the string "`Hello Java`" to a character array.
 - Writes the entire array to the file using the `write(char[], off, len)` method.
 - Here, `off = 0` and `len = chars.length` writes the full content.
-
- **`void write(String str, int off, int len)` – Writes a substring**

```
import java.io.*;

public class OutputStreamWriterWriteString {
    public static void main(String[] args) {
        String text = "OutputStreamWriter Example";
        try (OutputStreamWriter writer = new OutputStreamWriter(new FileOutputStream("output.txt")));
            writer.write(text, 0, text.length());
            System.out.println("String written to file.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```



Output:

```
String written to file.
```



File content (output.txt):

```
OutputStreamWriter Example
```



Explanation:

- This example writes the full string to the file.
 - You can also write part of the string by changing the **offset** and **length** values.
- **void flush() – Flushes the data forcibly to the file**

```
import java.io.*;  
  
public class OutputStreamWriterFlushExample {  
    public static void main(String[] args) {  
        try (OutputStreamWriter writer = new OutputStreamWriter(new FileOutputStream("Flush this data"));  
             writer.write("Flush this data");  
             writer.flush(); // Forces the buffer to be written immediately  
             System.out.println("Data flushed."));  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Output:

```
Data flushed.
```



File content (`output.txt`****):

Flush this data



Explanation:

- `flush()` makes sure all characters in the internal buffer are written to disk.
 - Useful when you want to make sure data is saved **before** the stream is closed or reused.
- `void close()` – Closes the writer and releases resources

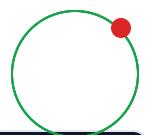
```
import java.io.*;  
  
public class OutputStreamWriterCloseExample {  
    public static void main(String[] args) {  
        OutputStreamWriter writer = null;  
        try {  
            writer = new OutputStreamWriter(new FileOutputStream("output.txt"));  
            writer.write("Closing Example");  
            writer.close(); // Manually closing  
            System.out.println("Stream closed successfully.");  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Output:

Stream closed successfully.



File content (`output.txt`****):





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



Java FileReader and FileWriter Classes

When working with files in Java, **FileReader** and **FileWriter** provide a convenient way to read and write text data using character streams. These classes are specifically designed for handling character files.

1. Class Hierarchy

```
java.lang.Object  
↳ java.io.Reader  
    ↳ java.io.FileReader
```

```
java.lang.Object  
↳ java.io.Writer  
    ↳ java.io.FileWriter
```

- **FileReader** extends **Reader**
- **FileWriter** extends **Writer**

2. FileReader Class

• Constructors



```
FileReader(String fileName)  
FileReader(File file)
```

These constructors open a file for reading. If the file doesn't exist, a **FileNotFoundException** is thrown.

- Example: Reading from a file using FileReader

```
import java.io.*;  
  
public class FileReaderExample {  
    public static void main(String[] args) {  
        try (FileReader reader = new FileReader("example.txt")) {  
            int ch;  
            while ((ch = reader.read()) != -1) {  
                System.out.print((char) ch); // Print each character one by one  
            }  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Output (if example.txt contains "Hello World"):

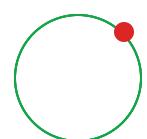
```
Hello World
```

- int read()

Reads a single character.

- int read(char[] cbuf, int offset, int length)

Reads characters into a portion of an array.



```
import java.io.*;

public class FileReaderReadArray {
    public static void main(String[] args) {
        char[] buffer = new char[100];
        try (FileReader reader = new FileReader("example.txt")) {
            int charsRead = reader.read(buffer, 0, buffer.length);
            System.out.println("Characters read: " + new String(buffer, 0, ch
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Output:

```
Characters read: Hello World
```

• void close()

Closes the reader. Automatically handled with try-with-resources.

```
import java.io.*;

public class FileReaderCloseExample {
    public static void main(String[] args) {
        FileReader reader = null;
        try {
            reader = new FileReader("example.txt");
            System.out.println("Reader opened.");
            reader.read();
            reader.close(); // manually closing
            System.out.println("Reader closed.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
}
```

Output:

```
Reader opened.  
Reader closed.
```



3. FileWriter Class

- Constructors

```
FileWriter(String fileName)  
FileWriter(String fileName, boolean append)  
FileWriter(File file)  
FileWriter(File file, boolean append)
```



These constructors create a file if it doesn't exist and optionally allow appending.

- Example: Writing to a file using FileWriter

```
import java.io.*;  
  
public class FileWriterExample {  
    public static void main(String[] args) {  
        try (FileWriter writer = new FileWriter("output.txt")) {  
            writer.write("Hello Java");  
            System.out.println("Data written successfully.");  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```



Output:

```
Data written successfully.
```



File **output.txt** will contain:

```
Hello Java
```



● void write(int c)

Writes a single character.

```
import java.io.*;

public class FileWriterWriteChar {
    public static void main(String[] args) {
        try (FileWriter writer = new FileWriter("output.txt")) {
            writer.write(65); // Writes 'A'
            System.out.println("Single character written.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```



Output:

```
Single character written.
```



File content:

```
A
```



- **void write(char[] cbuf, int off, int len)**

Writes a portion of a character array.

```
import java.io.*;  
  
public class FileWriterCharArray {  
    public static void main(String[] args) {  
        char[] data = "Welcome to Java".toCharArray();  
        try (FileWriter writer = new FileWriter("output.txt")) {  
            writer.write(data, 0, data.length);  
            System.out.println("Char array written.");  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Output:

```
Char array written.
```

- **void write(String str, int off, int len)**

Writes a portion of a string.

```
import java.io.*;  
  
public class FileWriterWriteString {  
    public static void main(String[] args) {  
        String msg = "Java FileWriter Demo";  
        try (FileWriter writer = new FileWriter("output.txt")) {  
            writer.write(msg, 5, msg.length() - 5); // Write "FileWriter Demo"  
            System.out.println("String written with offset.");  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
    }
}
```

Output:

```
String written with offset.
```



File content:

```
FileWriter Demo
```



• void flush()

Flushes the stream — forces any buffered output to be written.

```
import java.io.*;

public class FileWriterFlushExample {
    public static void main(String[] args) {
        try (FileWriter writer = new FileWriter("output.txt")) {
            writer.write("Flush Example");
            writer.flush(); // Force write
            System.out.println("Data flushed.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Output:

```
Data flushed.
```





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



Java BufferedReader and BufferedWriter Classes

In Java, **BufferedReader** and **BufferedWriter** are used for efficient reading and writing of text data. Unlike **FileReader** or **FileWriter**, they use an internal buffer to reduce the number of interactions with the disk, which makes them **much faster**.

Think of them like this:

Instead of reading or writing one character at a time (which is slow), they read or write **a whole chunk of data at once** and keep it in memory. This reduces the time it takes to access the file system again and again.

They are part of the **java.io** package and extend the **Reader** and **Writer** classes respectively.

1. Class Hierarchy

```
java.lang.Object
↳ java.io.Reader
    ↳ java.io.BufferedReader
```

```
java.lang.Object
↳ java.io.Writer
    ↳ java.io.BufferedWriter
```



- **BufferedReader** extends **Reader**
- **BufferedWriter** extends **Writer**

2. BufferedReader Class

- Constructor

```
BufferedReader(Reader reader)  
BufferedReader(Reader reader, int bufferSize)
```

You usually wrap a **BufferedReader** around another **Reader** like **FileReader**.

- Example: Reading line by line using **readLine()**

```
import java.io.*;  
  
public class BufferedReaderReadLineExample {  
    public static void main(String[] args) {  
        try (BufferedReader reader = new BufferedReader(new FileReader("input  
String line;  
        while ((line = reader.readLine()) != null) {  
            System.out.println("Read Line: " + line);  
        }  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```

Output:



```
Read Line: Hello  
Read Line: Java  
Read Line: World
```

Explanation:

- `readLine()` reads one line at a time and returns null when end of file is reached.
- It's commonly used when working with text data line-by-line.

• `int read()`

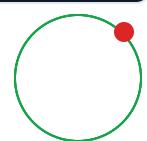
Reads a single character.

```
import java.io.*;  
  
public class BufferedReaderSingleChar {  
    public static void main(String[] args) {  
        try (BufferedReader reader = new BufferedReader(new FileReader("input"))){  
            int ch;  
            while ((ch = reader.read()) != -1) {  
                System.out.print((char) ch);  
            }  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Output:

```
Hello Java
```

• `int read(char[] cbuf, int off, int len)`



Reads characters into a portion of an array.

```
import java.io.*;

public class BufferedReaderCharArray {
    public static void main(String[] args) {
        char[] buffer = new char[50];
        try (BufferedReader reader = new BufferedReader(new FileReader("input.txt")));
            int numChars = reader.read(buffer, 0, buffer.length);
            System.out.println("Characters read: " + new String(buffer, 0, numChars));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Output:

```
Characters read: Hello Java
```

• **void close()**

Closes the reader and releases any resources. Automatically handled by try-with-resources.

```
import java.io.*;

public class BufferedReaderCloseExample {
    public static void main(String[] args) {
        BufferedReader reader = null;
        try {
            reader = new BufferedReader(new FileReader("input.txt"));
            System.out.println("First Line: " + reader.readLine());
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
```



```
        if (reader != null) {
            reader.close();
            System.out.println("Reader closed.");
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Output:

```
First Line: Hello  
Reader closed.
```

3. BufferedWriter Class

• Constructor

```
BufferedWriter(Writer writer)  
BufferedWriter(Writer writer, int bufferSize)
```

Usually wrapped around a `FileWriter`.

• `void write(String s)`

Writes a string to the buffer.

```
import java.io.*;  
  
public class BufferedWriterWriteString {  
    public static void main(String[] args) {  
        try (BufferedWriter writer = new BufferedWriter(new FileWriter("output")) {
```

```
        writer.write("BufferedWriter writing example");
        System.out.println("String written to file.");
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Output:

```
String written to file.
```



● void newLine()

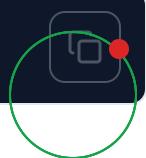
Writes a platform-specific newline.

```
import java.io.*;

public class BufferedWriterNewLine {
    public static void main(String[] args) {
        try (BufferedWriter writer = new BufferedWriter(new FileWriter("output.txt"))) {
            writer.write("Line 1");
            writer.newLine(); // Adds a newline
            writer.write("Line 2");
            System.out.println("Lines written with newLine().");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Output:

```
Lines written with newLine().
```



File **output.txt** content:

```
Line 1  
Line 2
```



- **void write(char[] cbuf, int off, int len)**

```
import java.io.*;  
  
public class BufferedWriterCharArray {  
    public static void main(String[] args) {  
        char[] chars = "Buffered Writer Test".toCharArray();  
        try (BufferedWriter writer = new BufferedWriter(new FileWriter("output.txt"))){  
            writer.write(chars, 0, chars.length);  
            System.out.println("Character array written to file.");  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```



Output:

```
Character array written to file.
```



- **void flush()**

Flushes the buffer, forcing any buffered output to be written.

```
import java.io.*;  
  
public class BufferedWriterFlushExample {  
    public static void main(String[] args) {  
        try (BufferedWriter writer = new BufferedWriter(new FileWriter("output.txt"))){  
            writer.write("Hello, World!");  
            writer.flush();  
            System.out.println("Buffer flushed successfully.");  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```



```
        writer.write("Flush this content");
        writer.flush();
        System.out.println("Data flushed manually.");
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Output:

```
Data flushed manually.
```



• void close()

Closes the writer. Automatically done in try-with-resources.

```
import java.io.*;

public class BufferedWriterCloseExample {
    public static void main(String[] args) {
        BufferedWriter writer = null;
        try {
            writer = new BufferedWriter(new FileWriter("output.txt"));
            writer.write("This will be saved on close");
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                if (writer != null) {
                    writer.close();
                    System.out.println("Writer closed.");
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Java Programming Handbook



Java StringReader and StringWriter Classes

In Java, **StringReader** and **StringWriter** are part of the **java.io** package. Think of them as virtual readers and writers that operate on in-memory strings rather than physical files. Just like **FileReader** reads from a file or **FileWriter** writes to a file, **StringReader** reads from a string and **StringWriter** writes into a string. This is especially helpful when you want to simulate file I/O or work with strings using stream-based APIs without touching the filesystem.

Imagine you're reading a story from a printed book — that's like using a **FileReader**. Now, think of reading the same story from a string saved in memory — that's where **StringReader** comes in. Similarly, instead of writing your thoughts to a notebook (like **FileWriter**), you write them into a digital sticky note stored in memory — that's what **StringWriter** does.

These classes extend **Reader** and **Writer** respectively and are used for reading from and writing to strings as if they were input/output streams.

1. Class Hierarchy

A dark rectangular overlay box with rounded corners. In the top right corner is a white square icon with a black square inside. In the bottom right corner is a green circular icon containing a white speech bubble with a red dot on top.

```
java.lang.Object
└ java.io.Reader
    └─ java.io.StringReader
```

```
java.lang.Object
└ java.io.Writer
    └─ java.io.StringWriter
```

- **StringReader** extends **Reader**
- **StringWriter** extends **Writer**

These are **character-based stream classes** that are especially useful for testing or when working with text in memory rather than with files.

2. StringReader Class

StringReader allows you to read characters from a **String** source.

● Constructor

```
StringReader(String s)
```

Creates a new **StringReader** using the specified string as the source.

● Example: Reading characters one by one

```
import java.io.*;

public class StringReaderExample {
    public static void main(String[] args) {
        String data = "Hello, Java!";
        // Create a StringReader from a string
```

```
try (StringReader reader = new StringReader(data)) {
    int character;
    // Read one character at a time
    while ((character = reader.read()) != -1) {
        System.out.print((char) character);
    }
} catch (IOException e) {
    e.printStackTrace();
}
}
```

Output:

```
Hello, Java!
```



Explanation:

- `read()` reads one character at a time from the string source.
- The loop continues until all characters are read.

● Example: Reading into a character array

```
import java.io.*;

public class StringReaderBufferExample {
    public static void main(String[] args) {
        String content = "Buffered Read";

        // Creating a character array buffer of size 20
        // This will store the characters read from StringReader
        char[] buffer = new char[20];

        try (StringReader reader = new StringReader(content)) {
            // Read characters into buffer starting at index 0
            int charsRead = reader.read(buffer, 0, content.length());
```



```
        System.out.println("Characters Read: " + new String(buffer, 0, ch
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Output:

Characters Read: Buffered Read



Explanation:

- A `char[]` buffer is created to hold the characters read from the string.
- The `read(char[] cbuf, int off, int len)` method fills the buffer from the string starting at the specified offset.
- This approach is more efficient when reading large strings or reading multiple characters at once.

• `close()`

Automatically handled with try-with-resources but can also be closed manually.

```
try (StringReader reader = new StringReader("Sample")) {
    // Operations
} catch (IOException e) {
    e.printStackTrace();
} // reader is automatically closed here
```



If not using try-with-resources:



```
StringReader reader = new StringReader("Sample");
reader.close(); // manually close
```

3. StringWriter Class

StringWriter allows you to write character data to a string buffer.

- **Constructor**

```
StringWriter()
StringWriter(int initialSize)
```

- **Example: Writing string content**

```
import java.io.*;

public class StringWriterExample {
    public static void main(String[] args) {
        // Creating a StringWriter
        try (StringWriter writer = new StringWriter()) {
            writer.write("Welcome to Java I/O"); // Writing to buffer
            String result = writer.toString(); // Get final string
            System.out.println("Written Content: " + result);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Output:

```
Written Content: Welcome to Java I/O
```

- Example: Append characters and strings

```
import java.io.*;  
  
public class StringWriterAppendExample {  
    public static void main(String[] args) {  
        try (StringWriter writer = new StringWriter()) {  
            writer.append('A'); // Appending a single character  
            writer.append("ppend Example"); // Appending a string  
            System.out.println("Appended Text: " + writer.toString());  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Output:

```
Appended Text: Append Example
```

- **getBuffer()**

Returns the underlying **StringBuffer** used by the writer.

```
import java.io.*;  
  
public class StringWriterBufferAccess {  
    public static void main(String[] args) {  
        try (StringWriter writer = new StringWriter()) {  
            writer.write("Buffer Access");  
            StringBuffer buffer = writer.getBuffer();  
            System.out.println("Buffer: " + buffer);  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Output:

Buffer: Buffer Access



• flush() and close()

`flush()` and `close()` are safe to call on a `StringWriter`, but they don't actually do anything. This is different from file-based streams like `FileWriter`, where `flush()` forces any buffered data to be written to the file and `close()` releases system resources. Since `StringWriter` writes to an in-memory buffer (a `StringBuffer`), there's no underlying resource to release or finalize. Therefore, calling `flush()` or `close()` won't change its behavior or output.

```
StringWriter writer = new StringWriter();
writer.write("Example");
writer.flush(); // Has no real effect but safe to call
writer.close(); // Also has no real effect
System.out.println("Output: " + writer.toString());
```



Output:

Output: Example

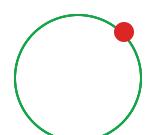


These methods are safe to call but do not affect the internal buffer.

4. When to Use StringReader and StringWriter

- `StringReader`:

- When you need to simulate file reading from a string.





Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



Q Java Programming Handbook



Java PrintWriter Class

In Java, the **PrintWriter** class is part of the **java.io** package and provides a convenient way to write formatted text to various output destinations such as files, memory buffers, or even network streams. It supports writing text using methods like **print()**, **println()**, and **printf()**, much like how we use **System.out**, but it's important not to confuse **PrintWriter** with **PrintStream** (used by **System.out**).

Unlike **FileWriter** or **BufferedWriter**, **PrintWriter** has the additional capability to handle formatted text and can automatically flush the stream after certain write operations, if configured to do so.

1. Class Hierarchy

```
java.lang.Object
  ↴ java.io.Writer
    └── java.io.PrintWriter
```



PrintWriter extends Writer

2. Constructors



```
PrintWriter(OutputStream out)
PrintWriter(OutputStream out, boolean autoFlush)
PrintWriter(Writer out)
PrintWriter(Writer out, boolean autoFlush)
PrintWriter(String fileName)
PrintWriter(File file)
```

- **autoFlush**: If true, the output buffer is flushed automatically whenever a `println`, `print`, `f`, or `format` method is called.

3. Key Methods

Method	Description
<code>print()</code>	Writes data without a newline
<code>println()</code>	Writes data followed by a newline
<code>printf()</code>	Writes formatted string (like C)
<code>flush()</code>	Forces any buffered output to be written
<code>close()</code>	Closes the stream

4. Examples

Example 1: Writing to a file

```
import java.io.*;

public class PrintWriterFileExample {
    public static void main(String[] args) {
        // Try-with-resources ensures automatic closing of PrintWriter
```

```
try (PrintWriter writer = new PrintWriter("output.txt")) {  
    writer.println("Hello from PrintWriter!");  
    writer.println(100); // writing integer  
    writer.println(true); // writing boolean  
} catch (IOException e) {  
    e.printStackTrace();  
}  
}
```

Output (in output.txt):

```
Hello from PrintWriter!  
100  
true
```

Example 2: Formatted output

```
import java.io.*;  
  
public class PrintWriterFormatExample {  
    public static void main(String[] args) {  
        // Printing formatted output to console  
        try (PrintWriter writer = new PrintWriter(System.out)) {  
            writer.printf("%-10s %10.2f\\n", "Price:", 25.50);  
            writer.printf("%-10s %10.2f\\n", "Discount:", 5.75);  
        }  
    }  
}
```

Output:

```
Price:      25.50  
Discount:   5.75
```

Example 3: Using autoFlush

```
import java.io.*;  
  
public class PrintWriterAutoFlushExample {  
    public static void main(String[] args) {  
        // Enabling autoFlush so flush is called after println  
        try (PrintWriter writer = new PrintWriter(System.out, true)) {  
            writer.println("This will flush automatically.");  
            writer.print("This won't add newline but will flush.");  
        }  
    }  
}
```

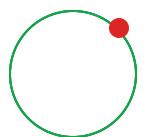


5. Why Use PrintWriter?

- Provides an easier and more readable API for writing text.
- Allows formatted output similar to `System.out.printf()`.
- Handles various data types automatically.
- Optional autoFlush for more control.
- Can write to files, memory buffers, sockets, etc.

6. Notes on flush() and close()

- `flush()` is used to push any buffered data to the destination.
- `close()` releases system resources and also flushes the stream.
- If `autoFlush` is enabled, you may not need to call `flush()` manually after `println()` or `printf()`.



7. Difference Between PrintWriter and PrintStream

While both `PrintWriter` and `PrintStream` are used for writing output and support methods like `print()` and `println()`, they are **not the same**. Here's a detailed comparison:

Feature	PrintWriter	PrintStream
Package	<code>java.io</code>	<code>java.io</code>
Inheritance	Extends <code>Writer</code> (character stream)	Extends <code>OutputStream</code> (byte stream)
Main Use	Writing text (characters)	Writing raw bytes and text
Encoding Support	Yes, supports character encoding (like UTF-8)	Limited encoding support
Exception Handling	Does not throw <code>IOException</code> on writing methods (must call <code>checkError()</code> to detect issues)	Same behavior (silently handles exceptions unless checked)
Output	Works with characters (e.g., writing to a file, console with text)	More suitable for binary data or console output
Example	<pre>PrintWriter pw = new PrintWriter("file.txt");</pre>	<pre>PrintStream ps = new PrintStream("file.txt");</pre>
Common Usage	Writing logs, reports, or formatted text to files or console	System output like <code>System.out</code> and binary logs

Important Note:

- `System.out` is a `PrintStream`, not a `PrintWriter`.
- If you want to write human-readable formatted text with better encoding handling (like UTF-8), use `PrintWriter`.
- If you're dealing with console output or binary output, `PrintStream` is more appropriate.



Java Tutorial – A Comprehensive Guide for Beginners

Introduction

Java is one of the most widely used and versatile programming languages in the world. From desktop applications to web services and mobile apps to enterprise systems, Java powers countless technologies and platforms.

This Java tutorial is designed for beginners who want to build a strong foundation in Java programming. Whether you're completely new to programming or transitioning from another language, this guide covers everything from core concepts to essential object-oriented principles and beyond.

If you're aiming to become a proficient Java developer, follow this structured path to master Java, step by step.

Prerequisites

Before you dive into Java, it's helpful (but not mandatory) to be familiar with:

- **Basic Computer Knowledge:** Understanding how software runs, what an IDE is, etc.
- **Logic Building:** Basic problem-solving and logical thinking.



- **Understanding of Programming Concepts** (optional): Knowing variables, loops, or functions from another language is a bonus.

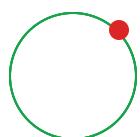
Master These Topics

Once you're ready, follow this structured list to learn Java from the ground up:

1. Introduction to Java
2. Java Fundamentals
3. Java Flow Control
4. Java Arrays
5. Java Methods
6. Java Object-Oriented Programming
7. Java Inner Classes
8. Static and Final Keywords
9. Java Exception Handling
10. Java Generics
11. Java Lambda Expressions
12. Java Collection Framework
13. Java Input/Output Streams
14. Java Reader/Writer

Introduction to Java

1. [Introduction to Java](#)
2. [Installing JDK on your computer](#)



[3. Java Comments](#)

Java Fundamentals

- [1. Java Variables and Literals](#)
- [2. Data Types in Java](#)
- [3. Operators in Java](#)
- [4. Java Basic Input and Output](#)
- [5. Java Expressions, Statements and Blocks](#)

Java Flow Control

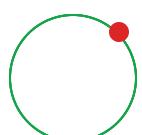
- [1. Java if else statement](#)
- [2. Java Ternary Operator](#)
- [3. Java For Loop](#)
- [4. Java while and do while loop](#)
- [5. Java continue and break statement](#)
- [6. Java Switch statement](#)

Java Arrays

- [1. Java Arrays](#)
- [2. Java Multidimensional Arrays](#)
- [3. Java Copy Arrays](#)

Java Methods

- [1. Methods in Java](#)



2. Parameter Passing in Java
3. Method Overloading
4. Variable Arguments
5. Java CommandLine Arguments
6. Java Recursive Method

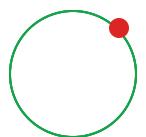
Java Object-Oriented Programming

1. Principles of Object-Oriented Programming
2. Java Class and Objects
3. Java Constructor
4. Encapsulation and Data Hiding
5. Inheritance in Java
6. Constructors in Inheritance
7. this Vs super Keyword
8. Method Overriding
9. Dynamic Method Dispatch
10. Polymorphism using Overloading and Overriding
11. Abstract Classes
12. Interfaces in Java

Java Inner Classes

1. Inner Classes in Java

Static and Final Keywords



1. Static Members and Static Blocks in Java
2. Final Members
3. Singleton Class

Java Exception Handling

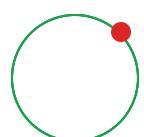
1. What are Exceptions
2. How to handle Exception
3. Try and Catch Block
4. Multiple and Nested Try Catch
5. Class Exception
6. Checked and Unchecked Exceptions
7. Throw Vs Throws
8. Finally Block
9. Try with Resources

Java Generics

1. Introduction to Generics
2. Defining Generic Class
3. Bounds on Generics
4. Java Generic Methods

Java Lambda Expressions

1. Introduction to Lambda Expressions
2. Parameters in Lambda Expressions



3. Capture in Lambda Expression

4. Method References

Java Collection Framework

1. Collections in Java

2. Understanding Collection Interface

3. Understanding List Interface in Java

4. Java ArrayList

5. Java LinkedList

6. Java Vector

7. Java Stack

8. Understanding Queue Interface in Java

9. Java PriorityQueue

10. Java ArrayDeque

11. Understanding Set Interface in Java

12. Java HashSet

13. Java LinkedHashSet

14. Java TreeSet

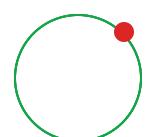
15. Understanding Map Interface in Java

16. Java HashMap

17. Java LinkedHashMap

18. Java TreeMap

19. Comparable Interface in Java



20. Comparator Interface in Java

Java Input/Output Streams

1. [What are Streams](#)
2. [InputStream and OutputStream Classes](#)
3. [FileInputStream and FileOutputStream](#)
4. [ByteArrayInputStream and ByteArrayOutputStream](#)
5. [ObjectInputStream and ObjectOutputStream](#)
6. [BufferedInputStream and BufferedOutputStream](#)
7. [Java PrintStream Class](#)

Java Reader/Writer

1. [Java File Class](#)
2. [Java Reader and Writer Class](#)
3. [InputStreamReader and OutputStreamWriter Class](#)



Pay Day Sale is Now Live! Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#).

6. [StringReader and StringWriter Class](#)



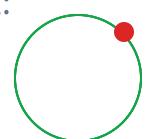
Java Programming Handbook



Key Features of Java

Java has remained popular for decades for good reason. Here's what makes it special:

- **Platform Independent:** Write once, run anywhere (thanks to the JVM).





 Pay Day Sale is Now Live!  Use the Coupon **PAYDAY** and Get Maximum Discount [Enrol Now!](#)



 Java Programming Handbook



Java Tutorial – A Comprehensive Guide for Beginners

Introduction

Java is one of the most widely used and versatile programming languages in the world. From desktop applications to web services and mobile apps to enterprise systems, Java powers countless technologies and platforms.

This Java tutorial is designed for beginners who want to build a strong foundation in Java programming. Whether you're completely new to programming or transitioning from another language, this guide covers everything from core concepts to essential object-oriented principles and beyond.

If you're aiming to become a proficient Java developer, follow this structured path to master Java, step by step.

Prerequisites

Before you dive into Java, it's helpful (but not mandatory) to be familiar with:

- **Basic Computer Knowledge:** Understanding how software runs, what an IDE is, etc.
- **Logic Building:** Basic problem-solving and logical thinking.



- **Understanding of Programming Concepts** (optional): Knowing variables, loops, or functions from another language is a bonus.

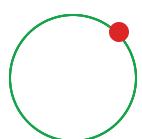
Master These Topics

Once you're ready, follow this structured list to learn Java from the ground up:

1. Introduction to Java
2. Java Fundamentals
3. Java Flow Control
4. Java Arrays
5. Java Methods
6. Java Object-Oriented Programming
7. Java Inner Classes
8. Static and Final Keywords
9. Java Exception Handling
10. Java Generics
11. Java Lambda Expressions
12. Java Collection Framework
13. Java Input/Output Streams
14. Java Reader/Writer

Introduction to Java

1. [Introduction to Java](#)
2. [Installing JDK on your computer](#)



[3. Java Comments](#)

Java Fundamentals

- [1. Java Variables and Literals](#)
- [2. Data Types in Java](#)
- [3. Operators in Java](#)
- [4. Java Basic Input and Output](#)
- [5. Java Expressions, Statements and Blocks](#)

Java Flow Control

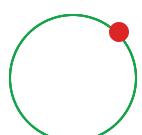
- [1. Java if else statement](#)
- [2. Java Ternary Operator](#)
- [3. Java For Loop](#)
- [4. Java while and do while loop](#)
- [5. Java continue and break statement](#)
- [6. Java Switch statement](#)

Java Arrays

- [1. Java Arrays](#)
- [2. Java Multidimensional Arrays](#)
- [3. Java Copy Arrays](#)

Java Methods

- [1. Methods in Java](#)



2. Parameter Passing in Java
3. Method Overloading
4. Variable Arguments
5. Java CommandLine Arguments
6. Java Recursive Method

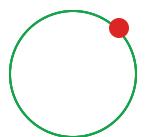
Java Object-Oriented Programming

1. Principles of Object-Oriented Programming
2. Java Class and Objects
3. Java Constructor
4. Encapsulation and Data Hiding
5. Inheritance in Java
6. Constructors in Inheritance
7. this Vs super Keyword
8. Method Overriding
9. Dynamic Method Dispatch
10. Polymorphism using Overloading and Overriding
11. Abstract Classes
12. Interfaces in Java

Java Inner Classes

1. Inner Classes in Java

Static and Final Keywords



1. Static Members and Static Blocks in Java
2. Final Members
3. Singleton Class

Java Exception Handling

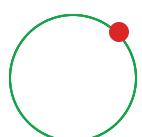
1. What are Exceptions
2. How to handle Exception
3. Try and Catch Block
4. Multiple and Nested Try Catch
5. Class Exception
6. Checked and Unchecked Exceptions
7. Throw Vs Throws
8. Finally Block
9. Try with Resources

Java Generics

1. Introduction to Generics
2. Defining Generic Class
3. Bounds on Generics
4. Java Generic Methods

Java Lambda Expressions

1. Introduction to Lambda Expressions
2. Parameters in Lambda Expressions



3. Capture in Lambda Expression

4. Method References

Java Collection Framework

1. Collections in Java

2. Understanding Collection Interface

3. Understanding List Interface in Java

4. Java ArrayList

5. Java LinkedList

6. Java Vector

7. Java Stack

8. Understanding Queue Interface in Java

9. Java PriorityQueue

10. Java ArrayDeque

11. Understanding Set Interface in Java

12. Java HashSet

13. Java LinkedHashSet

14. Java TreeSet

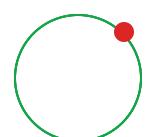
15. Understanding Map Interface in Java

16. Java HashMap

17. Java LinkedHashMap

18. Java TreeMap

19. Comparable Interface in Java



20. Comparator Interface in Java

Java Input/Output Streams

1. [What are Streams](#)
2. [InputStream and OutputStream Classes](#)
3. [FileInputStream and FileOutputStream](#)
4. [ByteArrayInputStream and ByteArrayOutputStream](#)
5. [ObjectInputStream and ObjectOutputStream](#)
6. [BufferedInputStream and BufferedOutputStream](#)
7. [Java PrintStream Class](#)

Java Reader/Writer

1. [Java File Class](#)
2. [Java Reader and Writer Class](#)
3. [InputStreamReader and OutputStreamWriter Class](#)
4. [FileReader and FileWriter Class](#)
5. [BufferedReader and BufferedWriter Class](#)
6. [StringReader and StringWriter Class](#)
7. [Java PrintWriter Class](#)

Key Features of Java

Java has remained popular for decades for good reason. Here's what makes it special:

- **Platform Independent:** Write once, run anywhere (thanks to the JVM).

